Contribution à l'étude du placement dynamique sur machines parallèles de type MIMD

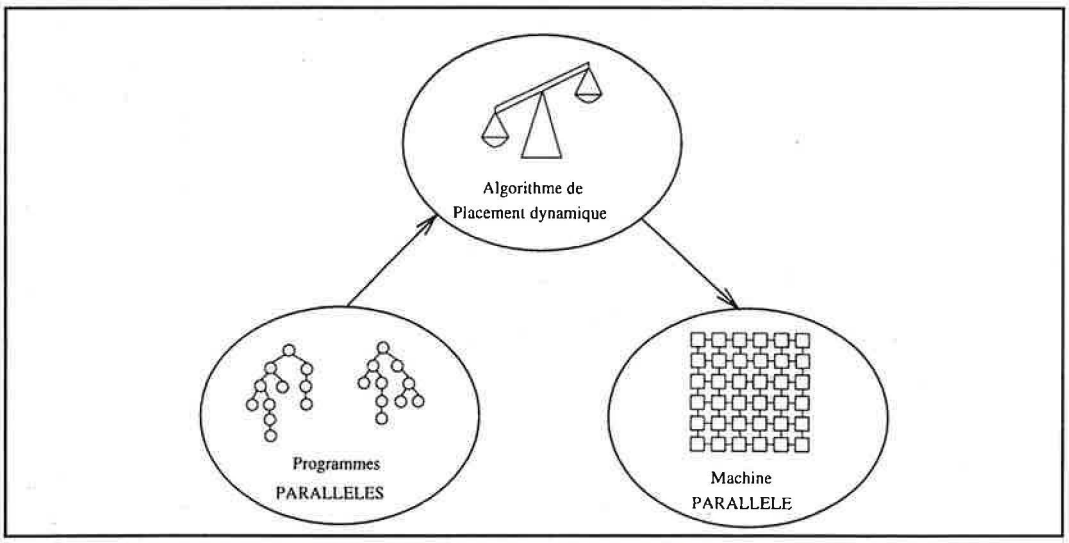
Graciela Roman-Alonso

UNIVERSITE
DE TECHNOLOGIE
DE COMPIEGNE

Mademoiselle ROMAN-ALONSO Graciela

Contribution à l'étude du placement dynamique sur machines parallèles de type MIMD

Thèse présentée
pour l'obtention du grade
de Docteur de l'UTC.



Soutenue le : 11 Juin 1997
Spécialité : Contrôle des Systèmes

D1024

CONTRIBUTION A L'ETUDE DU PLACEMENT DYNAMIQUE SUR MACHINES PARALLELES DE TYPE MIMD

Soutenue le 11 Juin 1997 devant le jury composé de :

MME SAYETTAT C. (Directeur de thèse)
M CARLIER J. (Président)
MME CORBEL A.
M MUNTEAN T. (Rapporteur)
MME ROUCAIROL C. (Rapporteur)
MM SAKHO I.
VAYSSADE M.

Remerciements

Le travail présenté dans ce document n'aurait pas pu être réalisé sans la participation d'un grand nombre de personnes qui m'ont apporté leur aide et leur soutien tant sur le plan scientifique que sur le plan humain.

D'abord mes remerciements aux institutions qui ont supporté mon travail, au CONACyT (consejo Nacional de Ciencia y Tecnologa) institution du gouvernement mexicain, qui, en collaboration avec la SFERE (Société Française d'Exportation des Ressources Educatives) en France m'ont accordé le support économique pour le bon déroulement de ma formation. Ma gratitude aux autorités de l'Universidad Autónoma Metropolitana-Iztapalapa de Mexico pour leur encouragement et leur appui dans le cadre du développement personnel des enseignants.

Ma reconnaissance aux laboratoires HEUDIASYC de l'Université de Technologie de Compiègne et Systèmes Coopératifs de l'Ecole des Mines de St-Etienne, dont j'ai pu partager l'ambiance amicale et scientifique, pour leur accueil et leur soutien.

Je tiens tout particulièrement à adresser mes remerciements à:

Mme. Claudette SAYETTAT, Professeur à l'Ecole des Mines de St-Etienne, pour m'avoir accueilli dans son équipe et pour avoir dirigé ce travail. Elle a porté un intérêt constant à mes travaux, sa manière de travailler extrêmement professionnelle m'a beaucoup appris.

Mme. Catherine ROUCAIROL, Professeur à l'Université de Versailles et à M. Traian MUNTEAN, Professeur à l'Université de Marseille II, pour avoir évalué ce travail et m'avoir conseillé et suggéré des améliorations.

Mme. Annie CORBEL, Maître Assistant à l'Ecole des Mines de St-Etienne, dont l'enthousiasme, les encouragements et l'intérêt sur ma recherche ont contribués à l'aboutissement de cette thèse.

M. Jacques CARLIER, Professeur à l'Université de Technologie de Compiègne, qui m'a fait l'honneur de présider le jury de cette thèse.

M. Ibrahima SAKHO, Ingénieur à l'Ecole des Mines de St-Etienne, qui a accepté de juger ce travail, et dont les remarques ont été d'une grande importance.

M. Michel VAYSSADE, Enseignant Chercheur à l'Université de Technologie de Compiègne, pour avoir participé à mon jury de thèse.

M. Roland JEGOU, Maître Assistant à l'Ecole des Mines de St-Etienne, pour le temps qu'il m'a consacré en lisant et faisant des remarques sur une partie de mon mémoire.

M. Gilles PAVARD, pour m'avoir introduit au fonctionnement général de SIMAD.

Nombreux sont mes amis des différentes spécialités qui ont collaboré à l'heureuse terminaison de mon travail et mémoire, il serait impossible de les citer tous mais impardonnable serait ne pas mentionner dans ce mémoire les plus proches.

Merci, Nadia KABACHI, Mihaela JUGANARU, Biche-Lien DOAN, Nathalie FARENC, Hervé LAMURE, Mohamad ALLOUCHE, Dionisio SUAREZ, Francisco RUIZ, Victor LARIOS, Simon et Amel PETIT-RENAUD, Felix RAMOS, Achour BELOUCIF, J.J. LEE et tous les amis de l'UTC et de l'EMSE.

Je tiens également à exprimer ma profonde gratitude pour leur soutien inconditionnel à Alejandro GUZMAN DE LEON ainsi que mes parents, frère et soeurs.

Dedico esta memoria a

Alejandro Guzmán De León,
mis padres Luisa y Baltazar,
mis hermanos Irma, Rosario y Alfredo,
mis abuelos Adalberto Román y Pascual Alonso,
mi querida tía Emma.

Table des matières

Table des figures

Introduction générale

L'évolution de la technologie a permis d'augmenter la puissance de calcul des ordinateurs. Dans un premier temps, les ordinateurs séquentiels ont représenté un grand pas fait par l'homme pour gagner du temps dans la résolution de certains problèmes. Cependant le développement de nombreuses disciplines, grosses consommatrices de temps de calcul (médecine, exploration, ingénierie, prévisions, secteur militaire, etc), a poussé les recherches de base sur le parallélisme, qui a pour objectif l'exécution simultanée des composantes d'un programme.

Le passage du traitement séquentiel au traitement parallèle a engendré une évolution tant au niveau matériel qu'au niveau logiciel. Plusieurs domaines de recherche contribuent à cette évolution, par exemple les architectures, la parallélisation automatique des programmes séquentiels, la tolérance aux pannes, l'évaluation de performances, les systèmes d'exploitation, l'algorithmique parallèle, le placement des processus et les langages de programmation parallèles. L'objectif global est la recherche de nouvelles architectures et de logiciels qui permettent une utilisation conviviale et efficace.

Lorsque l'on veut faire exécuter un programme, constitué de plusieurs processus, sur un système parallèle composé de noeuds de traitement reliés par un réseau de communication, un des problèmes qui se pose est le choix des sites d'exécution des processus. En particulier si le nombre de processus d'un programme parallèle est plus grand que le nombre de sites du système on doit décider sur quels noeuds on doit placer les processus pour exécuter le programme le plus vite possible.

Dans le cas où les caractéristiques des programmes sont connues avant l'exécution et le nombre de processus générés n'est pas trop important, nous pouvons utiliser des algorithmes exacts ou approximatifs de placement statique qui obtiennent respectivement, des placements optimaux ou acceptables des processus. Dans un autre cas, lorsque le nombre de processus présents pendant l'exécution d'un programme peut varier d'une exécution à une autre, il est préférable d'utiliser un algorithme qui alloue dynamiquement les noeuds aux processus.

Les algorithmes de placement dynamique ont deux tâches à réaliser. La première est celle de décider quand et sur quel noeud un processus va être placé. La deuxième concerne la collecte de l'information sur l'état de charge du système pour donner un appui à la prise des décisions. L'équilibrage de la charge est aussi parfois souhaité pour ne pas surcharger seulement quelques noeuds.

Lorsque l'on travaille avec des systèmes parallèles à mémoire distribuée où les noeuds sont incomplètement connectés, utilisant un réseau statique de communications (faiblement couplés), la récolte de l'information sur l'état global ou partiel de la charge peut pénaliser le temps total d'exécution. Les algorithmes qui distribuent la connaissance de l'état de charge du système sur tous les noeuds sont moins nombreux que ceux qui centralisent l'information.

Dans la plupart des algorithmes de placement dynamique les noeuds transmettent l'état de charge d'un seul noeud ou un vecteur contenant tous les noeuds du système à un autre noeud précis ou à tous les noeuds du système. La transmission d'un seul état de charge par message implique une génération importante de messages sur les liens de communication. Mais la transmission d'un vecteur contenant tous les états de charge des noeuds ne garantit pas la cohérence de l'information lorsque la charge des noeuds évolue de manière importante. La transmission d'un message contenant un sous-ensemble d'états de charge de plusieurs noeuds peut être une bonne solution car elle reduit le nombre de messages sur les liens et elle représente une information plus cohérente par rapport à l'état de charge réel de la machine. Cette solution, qui donne aux noeuds une connaissance locale de l'état de charge du système est peu envisagée, car elle pose le problème de l'actualisation du vecteur de charges.

Les performances apportées par les algorithmes de placement dynamique sont un sujet d'étude important. Plusieurs outils ont été conçus pour mesurer les performances de ces algorithmes exécutant certains programmes. L'existence d'une plateforme d'évaluation et de comparaison pour les nouveaux algorithmes de placement dynamique favorise la recherche sur l'amélioration des performances des systèmes parallèles.

Deux axes principaux constituent les travaux de cette thèse, d'une part la problématique du placement dynamique de processus et d'autre part les outils conçus pour étudier leurs performances. L'organisation de cette thèse est la suivante:

Le chapitre 1 est consacré à l'étude des algorithmes de placement. Nous présentons de manière générale la problématique du placement statique de processus et ensuite nous faisons un résumé des travaux proposés sur le placement dynamique. Ce cha-

pitre inclut aussi une classification basée sur le type d'information collectée dans ce dernier type d'algorithme.

Dans le chapitre 2 nous présentons les caractéristiques générales de quelques outils dédiés à la mesure des performances des systèmes parallèles. Le simulateur SIMAD, qui est la plateforme d'évaluation de notre travail est présenté de manière plus détaillée.

Le chapitre 3 est consacré à la proposition d'un ensemble de programmes pour tester les algorithmes de placement dynamique. Nous donnons d'abord un aperçu des types de programmes utilisés par quelques évaluateurs de performances. Ensuite nous présentons les instructions d'un langage synthétique proposé pour générer des groupes de programmes de test exécutés par SIMAD.

Un nouvel Algorithme Evolutif pour le placement dynamique de processus et l'équilibrage de la charge est proposé dans le chapitre 4. Nous présentons le type d'information possédée par chacun des noeuds et le fonctionnement de certains opérateurs évolutifs définis pour actualiser cette connaissance. Le comportement de cet algorithme est expliqué en détail ainsi que les paramètres de base qui règlent son comportement.

Une première partie du chapitre 5 présente une étude du comportement de l'algorithme Evolutif en faisant varier ses paramètres de base. La seconde partie montre une évaluation par comparaison de l'Algorithme Evolutif avec d'autres algorithmes exécutant les groupes de programmes de test.

Cette thèse se termine par une récapitulation des problèmes abordés et des apports effectués et par la présentation de quelques perspectives de nos travaux.

Chapitre 1

L'allocation de processus

Sommaire

Le besoin de minimiser les temps d'exécution des applications informatiques a provoqué l'évolution des architectures d'ordinateurs et des logiciels associés. Ceci débouche sur la conception de machines à grand nombre de processeurs (parallèles et massivement parallèles) dont le modèle de programmation est en relation avec le modèle d'exécution de la machine. Les modèles de programmation les plus courants

sont le parallélisme de données et le parallélisme de contrôle qui s'utilisent respectivement pour la programmation des machines de type SIMD ("Single Instruction Multiple Data") et MIMD ("Multiple Instruction Multiple data") [Mar93] [Eti94]. Dans le parallélisme de données, une même instruction est effectuée sur l'ensemble des données parallèles, ce qui correspond à une synchronisation implicite à chaque instruction. Un problème essentiel pour la classe de machine SIMD est la diffusion des instructions à partir du contrôleur central. Une solution a ce problème est l'utilisation des machines de type MIMD constituées d'un grand nombre de processeurs avec des mémoires physiquement distribuées. Dans ce cas, le modèle de programmation utilisé est le parallélisme de contrôle où diverses actions peuvent être effectuées sur des données différentes.

Les processus qui composent un programme parallèle peuvent s'exécuter en même temps sur différents sites de traitement. La façon de placer ces processus sur les noeuds d'une machine MIMD à mémoire distribuée a une influence sur le temps d'exécution du programme. Ce temps peut être significativement dégradé par une mauvaise utilisation des ressources (processeurs et réseau de communications).

Deux possibilités existent pour effectuer le placement des processus : soit au moment de la compilation et du chargement de l'application (allocation statique), soit lors de l'exécution de l'application (allocation dynamique). Dans le cas de l'allocation statique, on considère que certaines informations concernant les caractéristiques des programmes (durée d'exécution, ordre d'exécution, volume de communication, etc.) sont disponibles avant l'exécution afin de pouvoir guider le placement des processus sur les noeuds. Dans le cas de l'allocation dynamique, les stratégies de placement doivent disposer d'informations sur l'état actuel de la machine pour pouvoir décider du placement des processus au cours de l'exécution.

Ce chapitre présente dans une première partie un bref résumé des algorithmes de placement statique. La deuxième partie sera consacrée à une description plus élargie des méthodes pour le placement dynamique de processus.

1.1 Le placement statique

Un programme à placer de façon statique sur un ensemble de processeurs doit avoir un comportement connu ou estimé à l'avance de telle sorte que l'on puisse prédire et éventuellement déterminer la valeur de certains paramètres comme le coût des calculs effectués, le coût de transfert d'un processus, le volume et le coût des communications, les relations de communication, etc. Ces données peuvent être obtenues

par différents moyens, entre autres : par l'analyse faite à la compilation, par l'évaluation d'exécutions précédentes, par simulation ou par méthodes analytiques.

Dans le cadre statique un programme peut être modélisé comme un graphe non orienté dont les sommets sont des processus et les arêtes connectant deux noeuds représentent les communications entre les deux processus associés. On assigne à ces sommets et arêtes des nombres représentant respectivement les coûts de calcul et de communication. De la même façon la machine cible est modélisée par un graphe non orienté dont les sommets représentent les noeuds de la machine et les arêtes connectant deux sommets représentent les liaisons bidirectionnelles entre les deux noeuds associés. Dans ce cas les arêtes peuvent être pondérés par le coût de communications entre noeuds et les sommets par les coûts d'exécution des noeuds (puissances de calcul).

Le placement d'un programme T composé de m tâches communicantes $\{t_1, t_2, \ ..., t_m\}$, sur une machine parallèle P avec n processeurs interconnectés $\{p_1, p_2, \ ..., p_n\}$, est équivalent à la recherche de toutes les applications de $T \longmapsto P$. Soit S l'ensemble des placements possibles du programme T sur la machine P. Pour comparer les performances des placements on utilise une fonction de coût $f : S \longmapsto \mathbb{R}$ qui associe un coût réel à chaque placement $s_i \in S$. Le placement qui a le coût minimal est considéré comme un placement optimal du programme T.

Il y a clairement n^m placements (n choix pour t_1, n choix pour t_2, ... , n choix pour t_m), il est donc illusoire de tester tous ces placements lorsque le progamme ou la machine a un grand nombre d'éléments.
Malheureusement, le problème du placement étant NP-difficile [CC88], [GJ79] les seules méthodes garantissant l'optimalité de la solution nécessitent, dans le plus mauvais cas, d'explorer entièrement l'espace des solutions. Ces algorithmes ne peuvent être envisagés pratiquement que pour des problèmes de petites tailles.

Dans des cas restreints on arrive à réduire le problème de placement à un problème polynomial. Le travail fait dans [Lo88a], par exemple, présente un algorithme qui trouve la solution optimale en un temps polynomial. Les contraints considérées sont: la même durée d'exécution des processus et un nombre de processus inférieur à deux fois le nombre de processeurs dans la machine, plaçant, au maximum, deux processus par processeur.

Les caractéristiques du placement optimal que l'on désire obtenir dépendent de la fonction de coût utilisée. La fonction de coût d'un placement peut être déterminée selon les caractéristiques des programmes et du modèle d'architecture choisie.

La spécification de plusieurs fonctions de coût a été abordée dans des nombreux travaux [MLT82], [ST85], [LA87], [AP88], [MT91a]. Nous avons, par exemple, la fonction de coût ci-desous qui minimise le temps total d'exécution d'un programme exécuté sur un système avec des processeurs identiques reliés par un bus partagé.

$$f = \sum_{\substack{i,j \\ i<j}} C_{ij} Y_{ij}$$

avec
$i, j \in T$ ensemble de tâches
C_{ij} = coût de communication entre la tâche i et la tâche j
$Y_{ij} = 1$ si i et j sont sur des processeurs différents
$Y_{ij} = 0$ si i et j sont sur le même processeur

Les algorithmes de resolution du problème de placement proposés sont : soit des méthodes exactes qui permettent un placement optimal des processus sur les processeurs, soit des méthodes approximatives qui trouvent des placements acceptables (proches du placement optimal) dans un temps polynomial. Ces deux types de méthode sont utiles lorsqu'on a des systèmes spécialisés exécutant périodiquement des programmes au comportement connu, ils pourraient être incorporés dans un compilateur-linkeur-chargeur automatique du système parallèle [WM93].

1.1.1 Les algorithmes exacts

Les algorithmes exacts permettent de trouver au moins une solution optimale de placement. Des méthodes provenant de la théorie des graphes et de la programmation mathématique ont été utilisées pour proposer des algorithmes de résolution.

Quelques approches [ST85], [Bok81a] dans la théorie des graphes, ont été proposés pour trouver un homomorphisme faible entre le graphe représentant le programme et le graphe représentant le système parallèle. Dans ce cas le but est de placer deux processus voisins sur deux noeuds connectés par un lien physique. Une autre modélisation utilise un seul graphe contenant des sommets pour représenter les processus du programme et des sommets pour représenter les noeuds de la machine [Lo84], [Lo88b] [Sto77]. Les arcs du graphe relient un processus avec un noeud si le processus est exécuté sur le noeud, et relient un processus avec un autre processus s'ils font des communications, chaque arc étant pondéré par un coût. Le problème est de trouver la partition de coupe minimale du graphe, à chaque partition correspond un noeud avec un ensemble de processus qui seront exécutés sur celui-ci.

La programmation mathématique peut aussi être utilisée pour modéliser le problème du placement. A partir d'une fonction coût et d'un certain nombre de contraintes, on cherche à minimiser cette fonction coût en respectant ces contraintes, par des algorithmes sortant : de la programmation dynamique [Bok81b], des procédures par séparation et évaluation (branch-and-bound) [MLT82], [Sin87] [CY90], du A^* [ST85], etc.

Les algorithmes de séparation et d'évaluation appartiennent aux méthodes énumératives qui essayent d'évaluer une partie de toutes les solutions possibles. Plusieurs travaux se basent sur cette technique utilisant un arbre de recherche de solutions exploré de différentes manières [Sin87], [ST85], [CY90], [MM89](Branch&Bound). Ces algorithmes peuvent faire l'exploration jusqu'à trouver une solution satisfaisante mais dans le pire cas la complexité est exponentielle.

1.1.2 Les algorithmes approximatifs

Lorsque la taille du problème devient importante on utilise des placements approchés trouvés dans des temps raisonnables. Les solutions obtenues sont moins coûteuses en temps et en espace car les heuristiques utilisées n'examinent qu'une partie de l'espace total des solutions. Deux types d'algorithmes heuristiques qui conduisent assez rapidement à une solution approchée sont les algorithmes gloutons et les algorithmes itératifs.

Les algorithmes gloutons partent d'une solution de placement partielle (incomplète) pour ensuite construire petit à petit une solution finale complète. Le placement d'un processus p_j se fait en fonction des processus $p_1, p_2, ..., p_{j-1}$ déjà placés. Dans cette technique on ne peut pas remettre en cause un choix précédemment établi, c'est pour cette raison que le résultat n'est pas très performant. Quelques exemples proposés se trouvent dans [AP88].

D'autre part les algorithmes itératifs partent d'une solution initiale complète et cherchent ensuite à l'améliorer en faisant des permutations de processus jusqu'à obtenir une solution acceptable. La solution de départ n'est pas forcement une bonne solution, elle pourrait être obtenue de manière aléatoire ou bien par un algorithme glouton. Les améliorations se font en prenant en compte une fonction de coût associé à chaque solution.
Un algorithme itératif qui utilise une sorte de base de données pour garder les derniers mouvements effectués interdisant toute tentative de marche arrière sur une solution déjà explorée est la recherche Tabou [Len93].

Il existe d'autres techniques qui fonctionnent comme les algorithmes itératifs mais avec d'autres critères pour le choix de nouvelles solutions. Le recuit simulé, provenant de la thermodynamique appliquée à l'obtention de structures métalliques régulières, a été utilisé pour résoudre le problème du placement statique de processus [BM88], [US90]. Cette technique fonctionne comme un algorithme itératif sauf qu'elle peut accepter (sous une certaine probabilité qui décroît lors du déroulement de l'algorithme) des placements "moins bons" pour éviter d'être piegé dans un minimum local.

Les algorithmes génétiques [Gol89],[Ste93],[HAR94],[MT91b] sont basés sur la théorie de l'évolution des espèces. Etant donnée une population initiale de taille N, seulement les individus les plus forts pourront survivre (avec une certaine probabilité) et se reproduiront pour former de meilleures générations. Ces algorithmes on montré de bonnes performances par rapport aux algorithmes traditionnels d'optimisation dans la résolution de problèmes qui ont un espace de recherche multimodal [TBAM96]. Les différences principales entre les algorithmes génétiques et les méthodes traditionnelles sont l'utilisation d'une codification des paramètres du problème, la recherche dans une population de points (solutions admissibles) et l'évaluation d'une fonction objectif qui est plus générale qu'une fonction coût.
Quelques versions séquentielles et parallèles d'algorithme génétique [SG87], [ME90], [MT91b], [Neu91], [HAR94] ont été utilisées pour résoudre le problème du placement. Du côté séquentiel le travail fait dans [HAR94] utilise un graphe orienté pour représenter l'ordonnancement d'un ensemble des processus d'un programme. Un placement est trouvé en respectant les relations de précédence et tous les processus s'exécutent avec un délai minimum. Un individu a la forme de plusieurs listes de processus ordonnés et le nombre de listes est déterminé par le nombre de processeurs dans le système. La figure 1.1 montre le graphe d'un programme et un exemple de la codification de deux individus fournissant chacun un placement de 8 processus sur 2 processeurs. Les opérateurs de croisement et de mutation prennent en compte les relations de précédence des tâches pour générer des individus valides.

Le travail fait dans [MT91b] utilise un algorithme génétique parallèle de type diffusion (un individu par processeur) pour trouver le placement des processus d'un programme qui est défini comme un graphe sans circuit dont les noeuds sont les processus et les arcs les relations de communication entre les processus. Puisqu'il n'y a pas d'ordre à respecter, la codification d'un individu est représentée par une chaîne de M numéros qui identifient les processeurs (M égal au nombre de processus à placer). Un processeur dont le numéro est situé dans la position i devra exécuter

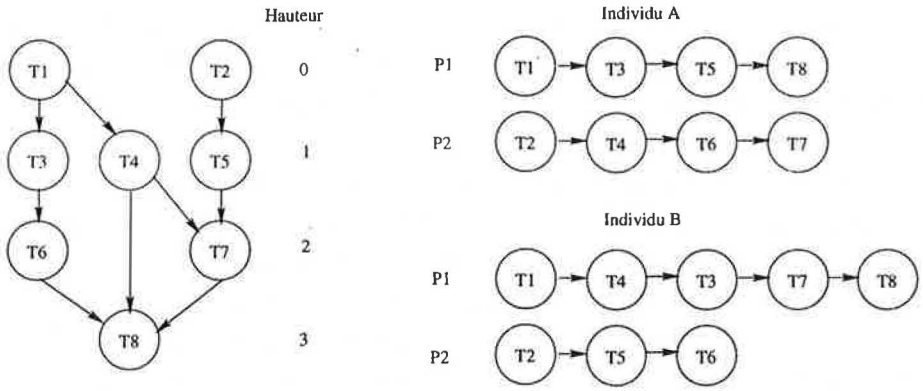


FIG. 1.1 - *Graphe d'un programme et codification de deux individus représentant des placements sur deux processeurs*

le processus i du programme. L'opérateur de croisement prend deux individus de la population, il choisit aléatoirement un point de croisement et il combine les différentes parties des parents pour former deux individus nouveaux, comme indiqué sur la figure 1.2. L'opérateur de mutation est unaire, il choisit aléatoirement une position de la chaîne et change au hasard le processeur qui exécutera le processus.

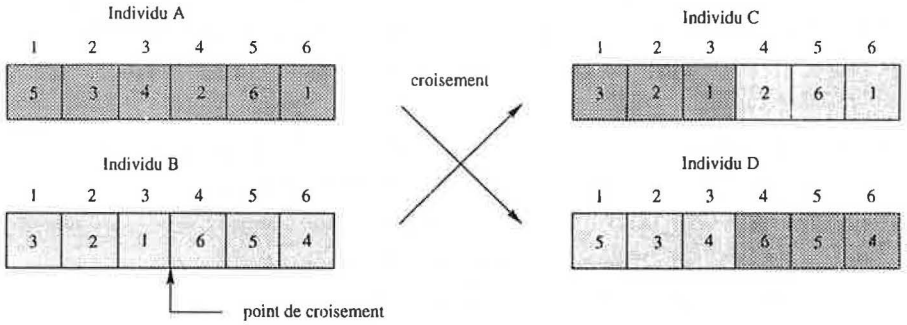


FIG. 1.2 - *Le croisement (crossover) de deux individus*

Cet algorithme est très intéressant, en effet son comportement évolutif permet de reconstruire de meilleurs individus (grâce à des échanges avec le voisinage). Il a inspiré la définition de certains opérateurs appliqués pour actualiser l'information partielle des noeuds dans le contexte de nos travaux sur le placement dynamique de processus [RAS94]. Cet algorithme sera présente plus en détail dans le chapitre 4.

Malgré la contribution importante des algorithmes d'allocation statique, leur utilisation est limitée par le temps de calcul qu'ils prennent pour trouver la solution. Il

faut aussi connaître à l'avance une estimation du comportement du programme par rapport à certains paramètres et dans beaucoup de cas, une description statique du système parallèle sur lequel on exécutera l'application.

Cette description statique fait par exemple défaut si différents utilisateurs accèdent simultanément aux ressources d'une seule machine. Le nombre de processus en exécution est alors imprévisible et leurs durées d'exécution et d'autres paramètres ne peuvent plus être connus avec précision. D'où la deuxième approche qui choisit le site d'exécution de chaque processus sur la base de l'état actuel du système. Puisque le manque d'information rend difficile la construction de solutions optimales, des recherches heuristiques ont été proposées. On appelle ce deuxième type de choix l'allocation dynamique de processus, plusieurs méthodes ont été proposées et seront présentées dans le paragraphe suivant.

1.2 L'allocation dynamique de processus

Les algorithmes d'allocation dynamique sont utilisés lorsque le nombre de processus présents dans l'exécution d'un programme parallèle varie d'une exécution à une autre en fonction des données d'entrée ou lorsqu'une connaissance statique de la machine n'est pas disponible. Les programmes basés sur une programmation objets ou acteurs (possédant des graphes de création et de communication qui évoluent dynamiquement) sont un exemple de programmes pour lesquels une allocation statique n'est pas possible. La tâche des algorithmes d'allocation dynamique est de décider en cours d'exécution d'un programme si les processus créés dynamiquement seront exécutés localement ou bien s'ils seront transférés vers d'autres noeuds. Normalement le transfert d'un processus est justifié par l'amélioration des performances du système.

Les instants pour décider du transfert ou de l'exécution locale des processus peuvent être soit à la création d'un processus, soit périodiquement ou lorsqu'on détecte une variation considérable de la charge sur les noeuds. On appelle ces deux types d'allocation respectivement le placement et la migration de processus.

1.2.1 Placement et migration

Dans le cas du placement, l'algorithme désigne, au moment de la création d'un processus, un noeud pour le faire exécuter intégralement. Le noeud choisi peut être le noeud où la création a été demandée, ou bien un noeud distant. Ce mécanisme est

également connu comme le placement définitif des processus (comportement non-préemptif).
Dans le cas de la migration, l'algorithme peut décider de transférer ou non un processus en cours d'exécution. Si c'est le cas, l'exécution d'un processus est interrompue et ensuite poursuivie sur un autre noeud (comportement préemptif) après avoir transféré le contexte d'exécution. Les deux problèmes les plus importants à résoudre sont de déterminer quels processus sont les plus appropriés pour être migrés et quels noeuds seraient les meilleurs récepteurs de processus de telle sorte que les performances du système s'améliorent. La migration de processus demande une évaluation préliminaire pour déterminer les gains qu'on pourrait obtenir si on déplace le processus en cours d'exécution ou bien s'il est préférable de le laisser finir son exécution dans le site où il se trouve.

Après une étude approfondie et une comparaison entre le placement et la migration de processus, une des conclusions présentées dans [BSS91] est que la migration apporte des bénéfices seulement dans des cas très particuliers (lorsqu'on migre des processus avec une durée de vie très longue par exemple). Dans [Fol96] la migration peut être liée aussi aux pannes des processeurs. Dans [EKMT94] l'utilisation de la migration est conseillée après avoir fait du placement dynamique des processus au moment de leurs créations. D'autres travaux [EKMT94] [Ell94] ont étudié les actions à réaliser pour rendre possible la migration transparente des processus sur des machines massivement parallèles, sans détériorer le temps de réponse du système. Dans ce cas, la migration a donné des bons résultats lorsque les temps d'exécution et d'inter-création des processus sont variables.

Les algorithmes d'allocation dynamique (utilisant le placement ou la migration de processus) considèrent le partage et l'équilibrage de la charge comme deux techniques pour l'amélioration des performances du système. Le partage de charge permet de transférer une partie de la charge d'un processeur surchargé vers d'autres processeurs. Ceci est fait lorsque la charge de travail locale à chaque processeur dépasse un seuil donné, ou bien, comme il est montré dans [Jég96], un processus est migré vers un noeud **Y** lorsque le processus doit utiliser les données existantes sur le noeud **Y**.

Quant à l'équilibrage ("load-balancing"), le but principal est qu'à tout instant la charge des processeurs soit équilibrée pour l'ensemble du système. Le placement ou la migration de processus ont lieu chaque fois que les conditions globales (vis à vis du nombre de processus en exécution sur chaque processeur) changent (création ou terminaison des processus). Ces techniques pourrait être appliquées pour aboutir à l'équilibrage global pendant l'exécution d'un programme.

1.2.2 Quantification de la charge locale d'un noeud

Dans la plus grande partie des travaux on suppose que l'état de charge de chaque processeur peut être représenté par une valeur appelée l'indicateur de charge. Les indicateurs prennent souvent en compte la longueur de la queue CPU qui contient l'ensemble des processus en état d'attente ou en cours d'exécution sur le processeur.

Pour certaines applications dont le nombre de processus varie considérablement l'indicateur de charge qui dépend du nombre de processus n'est pas très approprié car l'état de charge peut varier entre le moment de mesure et le moment de la décision. Dans ce cas une valeur moyenne du nombre de processus sur un intervalle de temps donné pourrait fournir de meilleurs résultats. Dans [TL89] la charge est représentée par une valeur moyenne des N derniers états de charge stockés dans un tableau de taille N.

Une autre façon de mesurer la charge est l'utilisation de seuils définissant des états de charge du noeud [Zho88], [LK87]. Cette solution est plus facilement utilisable lorsqu'il est difficile d'établir une métrique unique valable pour tous les éléments du système. Un exemple qui fonctionnent avec des mécanismes à double seuil de charge sera présenté dans la section 1.4.

Le volume de mémoire occupé dans un noeud peut aussi être utile pour la mesure de la charge. Cela pourrait éviter la distribution de processus sur des noeuds qui n'ont pas de place pour les faire exécuter. Pour les applications où le nombre d'opérations d'entrées-sorties est important on peut ajouter la longueur de la queue qui administre les entrées-sorties disque.

1.3 Structure générale des algorithmes d'allocation dynamique

Un algorithme d'allocation dynamique de processus fait partie du système d'exploitation d'une machine parallèle et pour l'implanter chaque noeud doit faire certaines actions (identiques ou complémentaires) qui contribuent au fonctionnement global de l'algorithme. Comme il a été dit précédemment, la tâche d'un algorithme d'allocation dynamique est de prendre une décision à un moment donné pour déterminer s'il faut déplacer un processus ou bien s'il est mieux de l'exécuter localement. La décision peut être prise par un seul noeud dans la machine (décision centralisée) ou bien par plusieurs noeuds (décision distribuée). La réponse peut dépendre d'une certaine information sur l'état de charge global ou partiel de la machine, et même sur

l'état de charge d'un seul noeud [ELZ86]. Cette information est appelée "l'élément d'information" qui constitue un des éléments principaux de l'algorithme d'allocation dynamique.

L'autre élément nécessaire au fonctionnement d'un algorithme d'allocation dynamique est "l'élément de contrôle" qui comprend deux composantes :

- la politique de transfert qui se charge de décider si le placement d'un processus est souhaitable selon les conditions globales ou partielles du système
- la politique de localisation qui détermine le processeur destinataire du transfert d'un processus (dans le cas qu'il doit être transféré).

Les éléments d'information et de contrôle sont complémentaires et permettent de définir un algorithme de placement. Dans les deux sections suivantes on expliquera ces deux éléments séparément en présentant quelques travaux associés aux différents techniques de traitement des informations et de prise de décisions.

1.3.1 L'élément d'information : information globale

Deux questions importantes se posent à propos de l'élément d'information. Comment obtient-on l'état de charge des noeuds et comment estime-t-on un état global ou partiel du système? Pour répondre à ces questions, un processus (exécuté sur chacun des noeuds) est mis en marche, il est chargé d'échanger des informations avec d'autres noeuds dans le système. Cette étape de collecte d'informations est cruciale pour les performances de l'algorithme. Si la fréquence des échanges est très courte il y aura une surcharge dans les liens de communications et l'exécution de l'application sera affectée. D'autre part, si la fréquence est très longue les informations risquent de ne plus représenter les états de charge actuels au moment où l'estimation est faite. Une fois que les informations ont été collectées certains calculs sont faits (s'il y a différents indicateurs de charge) pour estimer une charge totale ou partielle du système, puis l'information obtenue est mise à disposition du processus de décision.

1.3.1.1 L'information globale centralisée

La collecte globale centralisée implique que chaque noeud envoie son état local de charge vers un noeud unique X de la machine. Un processus situé sur le noeud X se charge de recevoir l'information de tous les autres noeuds pour avoir une vision globale de l'état du système. Le noeud X est le plus souvent un noeud destiné à recevoir l'information et à diriger la distribution des processus sur le système. Si un processeur Y exécutant un processus d'un programme doit créer un nouveau

processus le noeud X doit indiquer le site de placement pour ce processus. Cette stratégie est simple à implanter, mais elle présente plusieurs inconvénients :

- Un possible goulot d'étranglement sur le noeud X. Le temps de réponse aux requêtes de placement risque de devenir long si le nombre de noeuds devient grand.
- La panne du noeud X interrompt l'exécution de tous les autres noeuds. Une solution est la duplication du noeud central pour prévenir un nombre limité de pannes.

Une solution qui réduit la possibilité d'un goulot d'étranglement est la hiérarchisation des envois de l'information. On trouve dans [JS96] une stratégie qui utilise un arbre recouvrant une grille de processeurs. Les états de charge sont envoyés à partir des processeurs feuilles vers la racine de l'arbre. Le noeud racine reçoit seulement les charges équivalentes aux sous-arbres qu'il génère pour ensuite déterminer où il y a des excès de charge.

On trouve dans [DR94] 3 exemples d'algorithmes centralisés qui équilibrent la charge de 3 noeuds (stations de travail) exécutant des applications du type "branch-and bound". Les processus de l'application sont mis dans des queues d'attente des noeuds selon une priorité basée sur la potentialité de génération d'autres processus. Dans les trois algorithmes l'équilibrage de la charge est fait lorsque le noeud central reçoit une requête de la part d'un noeud sans charge en lui demandant du travail. L'exécution de l'application est arrêtée et poursuivie une fois que le noeud central détermine quels sont les processus qui vont être répartis pour équilibrer la charge.

1.3.1.2 L'information globale distribuée

La collecte globale distribuée implique que chacun des noeuds possède une information globale sur l'état de charge du système et est responsable de la distribution des processus. Ce type de collecte peut se faire de trois façons :

- Un fonctionnement initial identique à celui d'une collecte globale centralisée; un seul noeud reçoit les états de charge des autres noeuds et une fois qu'il a une information globale il la diffuse sur tous les noeuds du système.
- La diffusion de l'état de charge de chaque noeud vers tous les autres noeuds dans le système. Cette solution permet à chacun des noeuds d'avoir une vision globale de l'état du système et de pouvoir déterminer les noeuds sur lesquels seront distribués les processus du programme. Le problème de la panne d'un

processus central n'existe plus, mais une limitation sévère peut exister lorsque le nombre de noeuds devient important. Ces stratégies à caractère distribuée sont plus utiles dans des systèmes ayant un dispositif rapide de diffusion.

- La hiérarchisation des envois de l'état de charge. Par exemple dans le cas d' un tore, les noeuds situés sur la même ligne horizontale forment un anneau. En utilisant un jeton qui circule sur chaque anneau les noeuds pourrait mettre à jour leur état de charge. Un deuxième niveau d'information peut être formé par un anneau vertical (sur une des colonnes du tore) qui se charge de mettre à jours les informations obtenues par chaque noeud horizontal [RVV92].

1.3.1.3 Les collectes hybrides

La combinaison des deux types de collectes d'information (centralisée et distribuée) a été proposée [RF94]. Des stratégies hybrides définissant plusieurs niveaux de regroupement des noeuds permettent d'utiliser un type différent de collecte dans chaque regroupement. Le travail fait dans [RF94] choisit une stratégie centralisée pour les noeuds appartenant au même regroupement. Les noeuds uniques responsables de l'information de chaque regroupement communiquent entre eux pour avoir une estimation des charges de tous les autres regroupements.

1.3.2 L'élément d'information : information partielle

Le type de collecte hybride décrit dans le point précédent est un premier exemple d'information partielle. Chaque noeud central qui collecte l'information d'un sous-ensemble de noeuds possède une information partielle de l'état du système.

Lorsque tous les noeuds ont une information partielle de l'état de charge du système, une telle information doit être représentative et suffisante pour décider où et quand faire les transferts des processus. Dans l'algorithme du Gradient [LK87] par exemple, chaque noeud prend une décision de placement en tenant compte de sa propre charge et de son information partielle constituée d'indicateurs qui signalent la direction où les processeurs sont les moins chargés.

1.3.2.1 Les stratégies d'envoi de l'information

Plusieurs stratégies sont utilisées pour faire arriver les états de charge aux noeuds destinataires. Quelques solutions sont :

- L'utilisation du voisinage
  La diffusion est implémentée en transférant les états de charge de proche en

proche vers les noeuds destinataires. Un noeud envoie une information globale (ou partielle) de l'état de charge à ses voisins, ceux-ci actualisent l'information en mettant à jour leur propre état local et renvoient le message à leurs voisins respectifs [LK87], [BBS95]. Cette solution doit prévenir la génération d'un nombre interminable de messages.

- Le jeton circulant
  Dans cette solution un jeton contenant l'état global du système circule entre tous les noeuds. Chaque noeud qui reçoit le jeton actualise l'information en mettant à jour son propre état de charge et renvoie ensuite le jeton à son successeur. Les facteurs à déterminer sont la vitesse de circulation et l'ordonnancement des noeuds pour avoir le jeton.

- Profiter des messages transmis par les processus du programme
  Les états de charge des noeuds ne sont pas transmis par des messages spéciaux. Les états sont ajoutés aux messages échangés entre les processus du programme. Cela ne produit pas un sur-coût en nombre de messages mais l'information n'est pas fiable lorsque les processus des applications ne communiquent pas fréquemment.

1.3.2.2 Les initiatives pour l'envoi et la demande de l'information

Le type d'initiative utilisé pour envoyer ou pour demander des informations concernant les états de charge, permet de distinguer deux groupes de stratégies d'allocation dynamique :

- Le premier groupe est constitué des stratégies permettant aux noeuds d'envoyer leur état de charge sur la base d'une décision locale.

- Le deuxième groupe contient les stratégies avec un comportement à la demande, considérant les noeuds comme des entités qui envoient leur état de charge seulement si on les leurs demande.

Les noeuds, ayant une autonomie pour l'envoi de l'information, peuvent décider de faire l'envoi de leur états aux moments suivants :

- à intervalles de temps réguliers
  La fréquence moyenne d'envoi est un paramètre très important. Elle ne doit pas générer trop de messages qui surchargent les communications.

- à chaque changement de l'état de charge
  Cette solution évite d'envoyer des messages contenant de l'information inutile si l'état de charge est toujours le même.

- à chaque fois qu'on dépasse un certain seuil
  L'avantage ici est que les seuils peuvent être adaptés en fonction de l'estimation de l'état de charge global.

Les noeuds, ayant une initiative à la demande, envoient leur état de charge lorsqu'il y a une demande d'information. De cette façon seules les informations utiles circulent sur le réseau. Mais l'inconvénient est que chaque obtention d'information a besoin de deux envois de messages sur le réseau : la demande et la réponse. Le noeud collecteur de l'information peut faire des demandes de manière périodique ou lorsqu'on a besoin de placer des processus. Dans [ELZ86] on utilise une variable **Maxpoll** (Maxpoll $\leq$ nombre total de noeuds) qui représente le nombre de noeuds qui vont être interrogées sur leur charge. La détermination des noeuds est fait de manière aléatoire.

La combinaison de ces deux types d'initiatives est possible. Dans un système les noeuds peuvent avoir une autonomie, aux moments de surcharge, mais aussi une initiative à la demande lorsqu'ils sont faiblement chargés. Le travail fait dans [RACS96b] (chapitre 4) montre un algorithme de placement dynamique qui utilise les deux initiatives, autonome et à la demande, pour implanter un comportement évolutif dans l'obtention des informations.

1.3.3 Le genre d'information collectée

Plusieurs travaux donnent une classification des algorithmes d'allocation dynamique en fonction des type d'élément d'information et de contrôle utilisés [BSS91], [Tal93], [JM93], [Tal95], [Pav93]. Pour contribuer à la différenciation de tels algorithmes nous présentons une autre classification selon le type d'information collectée :

- Chaque noeud de la machine peut envoyer son état de charge soit à un seul noeud spécifique [DR94], soit à tous les autres noeuds de la machine [BSS91], soit aux voisins directement connectés [JS96] [BBS95].

- Chaque noeud envoie à son voisinage un indicateur signalant la direction du plus court chemin vers un noeud faiblement chargé. La valeur de cet indicateur est égale à la longueur du chemin et l'actualisation se fait de proche en proche lorsqu'il y a un changement. L'indicateur d'un noeud est égale à zéro quant il est faiblement chargé. Dans cette catégorie on trouve les travaux [Tal93], [LK87] et [PTS88].

- Un noeud envoie un vecteur de taille **N** contenant l'état de charge global du système (**N** = nb. de noeuds dans le système) [CF95]. Le vecteur contient

les états de charge de tous les noeuds et il est construit : soit de manière centralisée après l'obtention de tous les états de charge des noeuds, soit de manière distribuée où chaque noeud reçoit un vecteur global et actualise sa propre charge (le jeton circulant par exemple).

- Un noeud envoie un vecteur de taille **M** contenant l'état de charge de M noeuds (information partielle où **M** $\leq$ nb. de noeuds dans le système). Le travail fait dans [BS85] utilise un vecteur ***L*** de taille ***l*** dont le premier élément est la charge locale du processeur et le reste sont les charges d'autres ***l - 1*** noeuds arbitraires. Ce travail utilise des systèmes dont toute paire de noeuds est connectée par un lien de communication et travaille de façon asynchronne. Les échanges faits entre noeuds pour actualiser leur vecteur d'information fonctionne suivant l'algorithme :

  - 1. Modifier sa propre charge sur le vecteur
  - 2. Choisir au hasard un numéro i, $1 \leq i \leq N$, *N=Nb. de noeuds dans le système*
  - 3. Envoyer la première moitié de son vecteur au noeud i

  Lorsqu'un noeud reçoit une moitié d'un vecteur :

  - 4. Il forme un nouveau vecteur de taille l (toujours constante) de la façon suivante :

$$L(i) \longrightarrow L(2i), \quad 1 \leq i \leq \frac{l}{2} - 1 \ (\textit{Les positions paires})$$
$$L_R(i) \longrightarrow L(2i+1), \quad 0 \leq i \leq \frac{l}{2} - 1 \ (\textit{Les positions impaires})$$
$$\textit{où } L_R \textit{ est la moitiée du vecteur reçu}$$

  Le choix du noeud adéquat pour transférer un processus est déterminé par le noeud le moins chargé dans le vecteur.

- On trouve dans [RACS96b] (chapitre 4) l'algorithme **Evolutif** pour le placement dynamique de processus qui permet l'envoi d'informations contenant : soit un seul état de charge, soit un vecteur de taille variable = M, avec les états de charge d'au maximum M noeuds (M $\leq$ Nb. noeuds dans le système). Le principe est qu'un noeud X puisse compter sur un sous-ensemble de noeuds pour transférer des processus tout en équilibrant les charges entre le noeud X et les noeuds dans le vecteur. Lorsque les charges ont dépassé un certain niveau, des opérateurs évolutifs sont appliqués pour reconstruire un nouveau vecteur après la combinaison des vecteurs des noeuds voisins. Ce type de regroupement de l'information réduit le nombre de messages circulant sur la machine; il faut d'abord équilibrer la charge d'un sous-groupe de noeuds, et seulement lorsque les charges sont équilibrés on cherche à renouveler l'information pour obtenir

un autre vecteur représentant le nouveau domaine de placement pour chaque noeud. La taille maximale des vecteurs est important et est étudié dans la section 5.2.1. L'algorithme **Evolutif** définit pour chaque noeud une initiative autonome pour envoyer au voisinage son état de charge local et une initiative à la demande pour obtenir les vecteurs des noeuds voisins (seulement si cela est nécessaire).

1.3.4 L'élément de contrôle

L'élément de contrôle ainsi que l'élément d'information sont la base du fonctionnement d'un algorithme d'allocation dynamique. L'élément de contrôle d'un noeud permet de répondre à deux questions : quand la charge locale d'un noeud doit elle varier (augmenter ou diminuer) pour améliorer les performances du système? et à quels noeuds peut on transférer une partie de la charge ou bien demander de la charge?. Ces deux questions sont traitées respectivement par les politiques de transfert et de localisation.

Le noeud qui décide quand l'état de charge de certains noeuds doit varier (cherchant le simple partage ou bien l'équilibrage) est déterminé selon la localisation de l'information sur l'état de charge. On peut dire alors que l'élément de contrôle est centralisé ou distribué selon qu'un seul noeud possède une information globale ou que l'information (globale ou partielle) est distribuée sur tous le noeuds.

Lorsque l'élément de contrôle est centralisé, un seul noeud dans le système a une vue globale sur l'état de charge. Il est donc capable de déterminer les noeuds surchargés et sous-chargés à un moment donné pour ensuite décider si un transfert est nécessaire pour améliorer les performances.

Dans le cas où l'élément de contrôle est distribué, chaque noeud est capable d'estimer une charge globale ou partielle et peut décider d'augmenter ou de diminuer sa propre charge. La plupart des stratégies ayant un élément de contrôle distribué se basent sur un seuil de charge. Un noeud a deux options dépendant du seuil actuel :

- Il peut décider de transférer sa charge si le seuil de charge actuel équivaut à une sur-charge
- Il peut décider de demander plus de charge si le seuil de charge actuel équivaut à une charge faible

Le seuil qui indique la surcharge peut être fixé ou bien calculé dynamiquement. L'algorithme **Evolutif** qu'on présentera dans le chapitre 4 définit plusieurs niveaux de

charge qui peuvent s'adapter durant l'exécution (section 5.2.3).

1.3.5 La décision de transfert

Si la politique de transfert utilisée décide de transférer quelques processus vers d'autres noeuds pour diminuer la charge locale, la politique de localisation doit chercher des noeuds moins chargés pour les placer. Cette stratégie est appliquée plus souvent pour faire des placements au lieu de migrations de processus. Le transfert ne peut se faire qu'au moment de la création des processus.

D'autre part, si la politique de transfert cherche à augmenter la charge locale d'un noeud en demandant de la charge à d'autres noeuds, la politique de localisation doit chercher des noeuds fortement chargés pour leur demander des processus. Si cette stratégie permet un mécanisme de migration de processus, un noeud pourra prendre les processus en cours d'exécution sur d'autres noeuds. Un exemple d'utilisation de ce type de transfert se trouve dans [KR91].

Ces deux types de transfert ont certaines limitations. Si l'on utilise des transferts pour donner de la charge, lorsque le système devient chargé, beaucoup de noeuds fortement chargés perdraient du temps à chercher sans succès des noeuds faiblement chargés. En utilisant seulement le transfert pour recevoir de la charge un problème peut se poser lorsque la charge globale devient petite, car le nombre de noeuds faiblement chargés augmente et inonde le système de demandes de travail.

La combinaison de ces deux types de transfert a été proposée entraînant globalement une rapide distribution des processus. Néanmoins il y a le risque d'obtenir une instabilité où les processus font des aller-retours inutiles entre noeuds.

1.3.6 Politiques de localisation et de transfert

En général la politique de localisation prend en compte l'information (partielle ou globale) sur l'état de charge du système pour décider vers quels noeuds ou de quels noeuds on peut envoyer ou recevoir de la charge. Dans ce cas on dit que la politique de localisation est conditionnelle mais si aucune information n'est pas prise en compte on dit que la politique de localisation se fait en aveugle (inconditionnellement).

La localisation aveugle a été implantée en faisant des choix au hasard ou bien par

rotation pour éviter de choisir toujours un même processeur. Le processeur récepteur d'un processus peut exécuter le processus localement ou éventuellement le transférer à un autre processeur [ELZ86].

Dans la politique de localisation conditionnelle un noeud doit prendre en compte l'information qu'il possède pour faire un choix. Les éléments de décision dépendent de l'objectif à accomplir (donner ou recevoir de la charge) ils pourrait être : un choix parmi les noeuds les moins chargés (si l'on cherche à donner de la charge), ou bien un choix parmi les noeuds les plus chargés (si l'on cherche à recevoir de la charge). Lorsque plusieurs noeuds sont acceptables, un choix aléatoire peut être utilisé.

Dans le cas où la politique de transfert cherche à donner de la charge à d'autres noeuds, un mécanisme de Refus-Acceptation de requêtes peut être implanté dans la politique de localisation. Cela permet aux noeuds choisis de refuser ou d'accepter le transfert d'un processus, avant que celui-ci ait lieu.

Dans l'élément de contrôle la politique de transfert détermine quand un processus doit être placé, en prenant en considération l'état de charge des processeurs et parfois certaines caractéristiques des processus.

La création et la terminaison des processus font évoluer l'état global de la machine. Si l'on cherche à équilibrer la charge, les décisions prises par la politique de transfert pour placer un processus dépendent de cette évolution. Pour le seul partage de la charge on peut prendre en compte uniquement les instants de création ou bien les instants de terminaison des processus .

Normalement les politiques de transfert sont basées sur un seuil de charge. Les processus sont placés sur un autre processeur uniquement lorsque la charge locale dépasse un certain seuil (seuil indiquant que le processeur est surchargé). Le seuil qui indique la surcharge peut être fixé ou bien calculé dynamiquement. Dans le cas d'un seuil fixé, la valeur optimum dépend de la charge moyenne du système global. Dans [PTS88] on trouve un mécanisme adaptatif pour la détermination dynamique du seuil. Chaque processeur calcule son nouveau seuil par rapport à la moyenne des seuils des autres processeurs.

On trouve dans [BSS91] les différentes façons de prendre en compte les caractéristiques des processus pour décider leur placement dans les systèmes repartis faiblement couplés. Lorsqu'on a les connaissances précises des caractéristiques des processus (durée d'exécution) on peut utiliser un filtrage manuel pour indiquer au système quel processus est candidat au transfert, son inconvénient est la non-transparence

pour l'utilisateur. D'autres critères sont utilisés pour déterminer quel processus sera transféré sur un autre processeur en prenant en compte le type de processus, ou l'estimation du temps d'exécution qu'il lui reste.

1.4 Un exemple

Dans cette section nous présentons un exemple d'algorithme utilisé pour réguler la charge d'une machine parallèle à mémoire distribuée. Il s'agit de l'algorithme du Gradient [LK87] qui possède un mécanisme à deux seuils pour équilibrer la charge des processeurs.

Dans cette méthode, chaque processeur peut avoir une charge qui varie dans le temps et qui peut franchir deux seuils. Le premier seuil détermine le passage d'un état de charge légère à un état de charge moyenne (ou vice versa) du processeur. Le deuxième seuil détermine le passage d'un état de charge moyenne à un état de charge chargé (ou vice versa) du processeur.

Quelques notions théoriques définies pour le fonctionnement de la méthode du Gradient sont :

- La distance **d** entre deux processeurs **i** et **j** est la longueur du plus court chemin entre **i** et **j**.
- Le diamètre d'une architecture parallèle **N** est la distance maximale entre deux noeuds de N :

$$\text{diamètre(N)} = \max\{d_{ij} \; \forall \; i, j \; \in \; N \; \}$$

- La porte g_i d'un processeur **i** est une fonction binaire (ouverte/fermée). g_i est ouverte si le noeud **i** est légèrement chargé, dans un autre cas g_i est fermée :

$$\begin{aligned} g_i \; &= \; 0 \quad \longrightarrow \text{la porte est ouverte} \\ g_i \; &= \; W_{max} \quad \longrightarrow \text{la porte est fermée} \\ \text{où } W_{max} \; &= \; \text{diamètre(N)} + 1 \end{aligned}$$

- La proximité W_i d'un noeud **i** est la distance minimale entre le noeud **i** et un noeud légèrement chargé dans la machine. S'il n'y a aucun noeud légèrement chargé $W_i \; = \; W_{max}$

$$W_i = \begin{cases} min\{d_{ik}\}, & ssi \; \exists \; k \mid g_k \; = \; 0 \\ W_{max}, & si \; \forall \; k \quad g_k \; = \; W_{max} \\ 0, & si \; le \; noeud \; i \; est \; legèrement \; chargé \end{cases}$$

- La pression propagée p_i d'un noeud **i** est déterminée par le minimum entre les valeurs de g_i et la pression propagée minimale des voisins directement connectés au noeud **i** plus un

$$p_i = min\{g_i,\ 1\ +\ min\{p_j\ sur\ j,\ où\ d_{ij}\ = 1\}\}$$

La migration des processus est faite à l'aide d'une surface de gradients qui est l'ensemble de proximités de tous les noeuds. Pour avoir une approximation de la surface de gradients on utilise la pression propagée des noeuds. Chacun des noeuds exécute l'algorithme suivant pour équilibrer sa charge.

```
{
    déterminer l'état de charge du noeud i
    si l'état de charge est égal à :
        {
            léger : p_i = 0
            moyen : p_i = 1 + min{p_j, | d_ij = 1}
                    si (p_i > W_max)  p_i = W_max
            chargé : p_i = 1 + min{p_j, | d_ij = 1}
                    si (p_i > W_max)  p_i = W_max
                    sinon si (min{p_j} < p_i)
                        transférer un processus au noeud j, | p_j est le minimum
        }
    envoyer p_i aux voisins du noeud i si p_i a changé
}
```

La méthode du Gradient est simple et donne des bons résultats. Cependant les valeurs des seuils doivent être calculés selon le taux d'utilisation de la machine pour éviter une saturation précoce ou bien une sous-utilisation des ressources. Cet algorithme sera utilisé dans le chapitre 5 où nous ferons une comparaison entre plusieurs algorithmes d'allocation dynamique.

1.5 Conclusion du chapitre

L'allocation de processus est un problème relatif à la recherche de performances sur les systèmes parallèles, que ce soit des réseaux de stations, des machines parallèles ou bien des systèmes plus grands qui comprennent ces deux types de technologies. Les méthodes de placement statique de processus sont performantes lorsqu'on exécute des applications qui suivent toujours un même comportement. Dans ce cas le nombre de processus est une constante connue et les coûts de calcul et de communication

entre processus sont estimés par des valeurs numériques. Un système parallèle est aussi traité de façon statique, quant au nombre de processeurs et à leur configuration. Le placement statique peut donner des solutions optimales d'exécution mais le prix à payer est le temps qui croît exponentiellement lorsque le nombre de processus et de processeurs augmente.
Les méthodes d'équilibrage dynamique de processus sont indispensables pour placer au mieux des applications parallèles avec les caractéristiques suivantes :

- le nombre de processus est inconnu et change en cours d'exécution
- on n'a pas d'estimation du comportement des processus par rapport à la communication, la mémoire et les calculs
- les caractéristiques de la machine varient (système multi-utilisateurs par exemple)

L'élément d'information et l'élément de contrôle definissent la structure générale des algorithmes d'allocation dynamique. L'élément de contrôle (centralisé ou distribué) permet de décider du transfert d'un processus et de choisir le site de transfert. L'élément d'information se charge de collecter l'information (partielle ou globale) sur laquelle l'algorithme peut se baser pour prendre ses décisions. Le type d'un algorithme d'allocation dynamique dépend des caractéristiques de ces deux éléments.

La plus grand partie des travaux sur l'allocation dynamique de processus utilisent des éléments d'information qui récoltent les états de charge provenant, soit de tous les noeuds de la machine, soit d'un sous-ensemble de noeuds, soit du voisinage. Une fois que l'information est collectée, l'élément de contrôle a une vue globale ou partielle de l'état de charge du système et permet de décider sur quels noeuds les processus doivent être placés pour équilibrer la charge globale.

Lorsque l'on travaille avec des machine parallèles de type MIMD à mémoire distribuée avec un grand nombre de processeurs fortement couplés et communicant uniquement par échange de messages, la stratégie pour récolter l'information a une grande influence sur les performances. Dans le cas de machines parallèles où la distance entre les processus intervient dans le choix du site de placement, une stratégie distribuée qui donne une information partielle à chaque noeud est une bonne option. Un exemple est la méthode du Gradient pour équilibrer la charge d'une grille de processeurs en ayant aussi une double fonction de routage des processus sur la grille [Tal93].

Dans le chapitre 4 nous proposons un nouvel algorithme de placement dynamique de processus de type approximatif, avec un élément de contrôle et d'information distribués. Chaque noeud X possède un sous-ensemble de processeurs avec lesquels

il peut partager sa charge de manière équitable. Ce sous-ensemble est connu comme la Solution de Placement (SP) du noeud qui est mise à jour grâce à l'application de certains opérateurs évolutifs.

Chapitre 2

Les outils de simulation

Sommaire

L'amélioration des performances des machines MIMD comportant un grand nombre de processeurs (interconnectés par des liens de communication fiables) permettra de faire un réel gain sur le temps total d'exécution des applications. Pour cela il est très utile de pouvoir compter sur des outils permettant d'étudier le fonctionnement et les performances de ces environnements sans avoir besoin de tout le matériel physique et logique réels.

Des outils ont permis d'évaluer le comportement des applications s'exécutant sur des systèmes parallèles et composées d'un nombre variable de processus lors de chaque exécution. Ces outils permettent de reproduire le fonctionnement d'un système parallèle, allouant aux processeurs les processus des programmes pour estimer et ensuite prédire les performances de l'exécution. Les résultats obtenus sont fournis au moyen de graphiques et de statistiques. De plus ils aident l'utilisateur lors du choix de paramètres importants.

La simulation utilise la réduction d'un problème complexe à un autre problème équivalent plus simple dont on est capable d'analyser le comportement. Pour évaluer les performances des systèmes parallèles, plusieurs auteurs utilisent des modélisation de systèmes informatiques différentes.

Ce chapitre commence par une description générale de quelques outils construits pour l'évaluation des performances des systèmes parallèles. Ensuite nous consacrerons le reste du chapitre à la description du fonctionnement de l'outil SIMAD [Yah94] construit pour tester et comparer des méthodes de placement dynamiques de processus et qui a été utilisé comme plateforme d'évaluation de notre travail [PRAS95, RACS96a].

2.1 Quelques travaux pour l'évaluation de performances de systèmes parallèles

Dans cette section nous présentons d'une manière générale quelques outils développés pour la mise au point des performances des systèmes parallèles. Ces outils

permettent d'étudier les performances des applications (qui peuvent être modélisées de différentes manières) utilisant un certain type d'algorithme de placement.

2.1.1 Le simulateur PARSEVAL

Les systèmes informatiques peuvent être modélisés par un réseau de files d'attente. PARSEVAL [RM91] est un simulateur distribué de réseaux de files d'attente, fonctionnant sur un réseau de tansputers. Le but est d'utiliser de façon optimale la configuration matérielle disponible pour améliorer les performances.

La modélisation d'un système à l'aide de réseaux de files d'attente permet de le décomposer en plusieurs activités indépendantes, communiquant par des messages (qui sont appelés "clients"). Chacune des activités est représentée par une file d'attente munie de son serveur. Chaque file est simulée par un processus et la simulation sera constituée de plusieurs processus s'exécutant en parallèle. Une file d'attente est définie par un ensemble de paramètres qui décrivent les façons de générer et de traiter les messages qui circulent dans le système (voir section 3.3.1).

Le simulateur PARSEVAL est composé de deux parties distinctes : une interface graphique exécutée sur l'ordinateur hôte (une station SUN) et un système de simulation exécuté sur un réseau de transputers.

L'interface graphique permet à l'utilisateur de décrire le modèle de réseau de files d'attente et de générer un code (OCCAM [INM88]) qui spécifie explicitement le placement des processus (placement statique) et qui sera par la suite exécuté par le système de simulation.

Le système de simulation est constitué d'un réseau de routage et d'un ensemble d'applications qui simulent les files d'attente. Le système de routage est un réseau en anneau, dont le nombre de noeuds peut être défini selon le matériel disponible et le réseau de files d'attente à simuler. Sur chacun des processeurs sont placés un routeur, des simulateurs de files d'attente et des sources de messages. Le placement optimal est obtenu lorsqu'un seul simulateur est placé sur un processeur (si le nombre de processeurs est suffisant).

Lorsque la simulation se termine (la date finale a été atteinte par tous les simulateurs des files d'attente), chacun d'entre eux envoie pour affichage les résultats statistiques suivants :

- le nombre de messages nuls envoyés (qui donnent des indications sur la synchronisation des noeuds)

- le nombre total de messages arrivés dans la file
- le nombre total de messages servis et envoyés
- la durée moyenne de service qui est la moyenne de tous les services
- la longueur moyenne de la file par unité de temps
- le temps moyen d'occupation d'un serveur par unité de temps

Le principal intérêt de ce travail a été la proposition d'un simulateur à événements discrets distribué basé sur le modèle de files d'attente pour étudier le comportement des systèmes informatiques. L'aspect étudié principalement est le comportement du système vis à vis de la génération et du traitement des messages plutôt que le placement des processus à proprement parler.

2.1.2 Une plateforme d'évaluation dévéloppée à l'UST de Lille

Le travail fait par F. HEMERY [Hem94] présente une plateforme d'évaluation qui donne à l'utilisateur la possibilité d'étudier le comportement d'une application parallèle en faisant varier son découpage. L'auteur considère qu'un algorithme de placement dynamique doit prendre en compte les caractéristiques des processus à placer pour mieux exploiter les ressources de la machine. La plateforme a les objectifs suivants :

- Permettre une évaluation des stratégies de placement
  Pour chacune des stratégies il est possible de déterminer les paramètres qui règlent son comportement.
- Déterminer les caractéristiques des processus utilisés
  La plateforme garde des informations sur le coût d'exécution des processus et sur les relations de communication entre eux. Ces informations sont ensuite mises dans le code de description de l'application pour permettre à la stratégie de placement de prendre en compte les caractéristiques des processus et ainsi faciliter leur placement.
- Aide au découpage de l'application
  La plateforme se charge d'établir un découpage de l'application en processus ou entités.

- Déterminer la topologie la mieux adaptée à l'application
  Plusieurs topologies de machines peuvent être testées, la plateforme pourra déterminer celle qui est la plus adaptée pour exécuter l'application.

L'ensemble de stratégies de placement implantées disposent d'un élément de contrôle distribué (chaque noeud décide du placement des entités à créer). Les algorithmes avec un élément de contrôle centralisé ne peuvent pas être utilisés dans ce modèle d'exécution. Les sites de placement ne sont pas remis en cause une fois que les entités commencent leur exécution (il n'y a pas de mécanisme de migration).

Le langage GENESE peremet la description synthétique de "Cacs" (Composant Actif de Communication), il a été défini pour pouvoir tracer les événements importants d'une exécution. Ce langage permet de synthétiser le comportement d'une application grâce à plusieurs instructions qui spécifient l'utilisation des ressources de communication et de calcul (voir section 3.3.2). L'intérêt est de pouvoir générer des charges dans le système sans avoir besoin d'expliciter les objectifs des utilisations des ressources.
Cette plateforme est surtout utile lorsqu'on s'intéresse à l'étude du comportement général des applications parallèles. Les performances sont étudiées en faisant varier certains paramètres qui règlent le comportement des algorithmes d'allocation dynamique avec un élément de contrôle distribué.

2.1.3 Le modèle pour l'analyse du parallélisme proposé par Kwiatkowski et Stromboni

Ce modèle pour l'analyse du parallélisme [SK93] est composé de trois autres modèles : le modèle de l'algorithme (ou application), le modèle de la machine (des communications et des unités de traitement parallèle) et le modèle de l'implémentation (directives de placement et d'exécution).

- Un algorithme (programme parallèle) est modélisé par un graphe orienté. Les sommets sont composés par des variables et des opérations associées (constituant les processus), et les relations de dépendance entre processus sont représentées graphiquement par les arcs du graphe de processus (voir section 3.2).

- Le modèle de la machine est aussi décrit par un graphe orienté contenant l'ensemble des processeurs (sites) et des liens de communication. Les sommets sont les processeurs et les arcs sortant d'un sommet spécifient des liaisons physiques unidirectionnelles vers les processeurs voisins. Ce graphe est décom-

posable par le regroupement des liens et le partitionnement des noeuds. Pour décrire le fonctionnement d'une machine il faut également préciser :

- Le modèle des communications qui décrit le déroulement des transferts entre sites au cours de l'exécution de l'algorithme :
  - routage
  - mémoire commune
  - réseau d'interconnexion reconfigurable
- Le modèle du traitement parallèle qui permet différents modes de calcul parallèle. Ces modes diffèrent par les contraintes de synchronisation, la condition de démarrage du calcul et la charge acceptée. Les 4 modes implémentés sont SIMD, SPMD, Dataflow et Systolique. Ce modèle permet aussi de spécifier une liste de variables dont le placement est imposé, et la liste de processeurs sur lesquels les variables restantes doivent être placées.

- Le modèle de l'implémentation traite le problème du placement des variables (et leurs opérations) sur les noeuds. Ils présentent deux stratégies distinctes de placement dans le modèle : l'initialisation de placement ou l'optimisation de placement.

  L'initialisation de placement se fait lors du chargement de l'algorithme ou lors de la compilation pour avoir un placement initial. Dans ce cadre les auteurs ont utilisé des stratégies de type "glouton" qui déterminent le placement d'une variable lors de la lecture de la description d'un algorithme en fonction des variables déjà placées. La performance de l'heuristique de placement est mesurée pour les calculs et les communications par l'indicateur de deséquilibrage de charge suivant :

  $$Deséquilibre = (charge\ max\ -\ charge\ min)\ /\ (2\ +\ charge\ moyenne)$$

  L'optimisation de placement est la seconde stratégie où, à partir d'un placement initial, on recherche le placement qui optimise une fonction critère reliée à l'équilibrage des charges des ressources parallèles. Une des heuristiques présentées est le recuit simulé [XH91].

Avec l'utilisation de ce modèle pour l'analyse du parallélisme on peut obtenir les résultats suivants :

- Taille mémoire nécessaire dans chaque site, et placement des processus
- Utilisation cumulée des liens et des noeuds, sous forme de rapport d'activités

- Temps d'exécution (en nombre de cycles)
- Performance de la parallélisation (accélération, efficacité, utilisation des sites et du réseau)

D'une manière générale ce modèle nous permet d'étudier les performances des systèmes exécutant des algorithmes qui peuvent être spécifiées sous la forme d'un graphe décomposé éventuellement en phases. Le type du placement des processus est statique et prend en compte l'ordre d'exécution des échanges et l'ordre de précédence des données.

2.1.4 ATHAPASCAN

Le système ATHAPASCAN [Chr94, Pa93] est un environnement d'exécution de programmes parallèles portables à destination de systèmes à grand nombre de processeurs. Son objectif est la régulation de la parallélisation d'applications initialement dans le domaine du calcul formel et des bibliothèques numériques de base. Les auteurs proposent un format d'écriture des programmes parallèles pour que leur découpage en sous-calculs soit adapté aux caractéristiques des machines cibles. Les principes sont les suivants :

- L'utilisation de directives générales (adaptables à toutes les machines parallèles) pour exprimer le découpage du programme. La portabilité des méthodes de découpage rend possible la portabilité du programme. Le découpage se fait à partir des caractéristiques de la machine et du problème.
- Le découpage peut être arrêté. Les critères d'arrêt du découpage se basent sur des fonctions de contrôle permettant de décider statiquement ou dynamiquement du choix du degré de découpage parallèle.

ATHAPASCAN est constitué de trois modules principaux : Un noyau exécutif parallèle (ATHAPASCAN -0), une aide à l'implantation de programmes parallèles (ATHAPASCAN -1) et un outil d'évaluation des performances et de débogage.

Une machine parallèle ou un réseau de stations supportant ATHAPASCAN-0 assure la portabilité du reste de l'environnement. Les seuls logiciels indispensables sont le langage C et une bibliothèque de communication par passage de messages (PVM-Parallel Virtual Machine ou MPI-MessagePassing Interface). Un programme en ATHAPASCAN est un graphe d'appel de procédures [CC94] et une machine ATHAPASCAN est un ensemble de serveurs supportant ces appels de procédures à distance (synchrone ou asynchrone) avec l'approche utilisé SAMD ("Single Application Multiple Data").

L'outil de placement ATHAPASCAN-1 calcule d'abord un placement statique du graphe de tâches (qui peut être connu statiquement). De là on obtient un ensemble de tables de chargement statique des tâches et une estimation du coût d'exécution du programme avec ces placements. Pour corriger de possibles imperfections du placement statique, on obtient une estimation approximative de l'état du système pour décider de la redistribution de la charge. Un autre axe de recherche est la régulation dynamique pour des programmes pour lesquelles on ne dispose pas des caractéristiques d'exécution.

Un autre outil dans ATHAPASCAN permet la mise au point des exécutions en permettant des débogages. Cet outil permet l'étude des exécutions en enregistrant un comportement initial du programme.

Un étude en cours de développement pour l'évaluation des stratégies de régulation de charge [CRP96], utilise le modèle de programmes ANDES ("Algorithms aNd DEScription") [Kit94] pour créer les applications en ATHAPASCAN et avoir une description quantitative d'applications parallèles. La finalité est de guider les choix d'une stratégie de répartition en fonction des caractéristiques globales du programme.

2.1.5 L'outil basé sur AIMS

L'outil présenté par Yan, Sarukkai et Mehra [YSM95] utilise une méthodologie basée sur AIMS ("Automated Instrumentation and Monitoring System"). Cet outil est construit pour mettre au point et pour prédire la performance de programmes à passage de messages sur des systèmes multiprocesseurs fortement ou faiblement couplés. Il n'est pas associé à une architecture ou à un système particulier et peut être porté sur des nouvelles plateformes physiques.

Il est possible d'utiliser plusieurs logiciels pour la mesure et l'analyse des performances. Le premier a comme entrée un programme (larges applications scientifiques en C ou en Fortran) et insère des commandes à des points sélectionnés qui indiquent les endroits de sauvegarde des données de performance. Durant l'exécution de ce programme une série d'événements est rassemblée dans un fichier de traces associé à une architecture. La perturbation de l'algorithme qui collectionne les événements est éliminée par un module de compensation d'intrusion. Ce dernier fichier compensé est l'entrée de plusieurs outils de post-exécution qui déterminent les caractéristiques de performance d'un programme, par visualisation, profil d'exécution, mesures dans les points spécifiques et modélisation. Le type de placement utilisé dans cette plate-

forme est statique. Les principaux avantages offerts par cet outil sont l'étude du comportement des applications utilisées dans des domaines scientifiques ainsi que la prédiction de leurs performances en utilisation réelle.

2.1.6 L'évaluateur de stratégies de régulation de charge pour le système PloSys

Le travail présenté par J. Briat, S. Kannat et E. Morel [BKM95] est une plateforme d'évaluation des stratégies de régulation dynamique de charge pour le système logique parallèle PloSys. Ce système exploite le parallélisme suggéré par la structure de l'arbre ET/OU qui est généré par l'évaluation d'un programme Prolog. Ce parallélisme permet l'évaluation simultanée de plusieurs branches de l'arbre en fonction des processeurs disponibles.

Chaque processeur possède un espace de travail formé par les alternatives en attente d'évaluation (vu comme un ensemble de processus). Lorsqu'un processeur termine l'exploration de son espace de travail il pourra acquérir un nouvel espace. La répartition de charge a lieu lorsqu'un processeur a beaucoup d'alternatives à évaluer et que d'autres processeurs se trouvent dans l'oisiveté.

L'environnement de cette plateforme est implanté sur un Meganode (composé par des transputers T800). Il est composé par trois types de serveurs.

- Le serveur travailleur (exécuté sur chaque processeur) est implanté par deux processus. Le premier est le simulateur qui modélise le moteur d'inférence de Prolog. Il traite les exécutions et les créations de processus. Le deuxième est destiné à estimer le temps d'oisiveté du processeur

- Le serveur gestionnaire des communications a comme tâche de gérer les messages de communications entre processeurs.

- Le serveur qui applique la fonction de régulation de charge est formé par trois processus. Le premier est un contrôleur central exécuté sur un processeur dédié (processeur de contrôle) qui a une connaissance globale de l'état de charge du système. Le contrôleur central garde une trace de l'exécution effectuée sur chacun des processeurs, lorsqu'il prend des décisions de régulation il envoie les commandes de régulation aux contrôleurs locaux des deux processeurs importateur-exportateur. Le deuxième processus est un contrôleur local qui traite les commandes de régulation provenant du contrôleur central. Une stratégie de sélection lui permet de choisir les processus à transférer. Le dernier

est un processus espion qui se charge d'envoyer périodiquement au contrôleur central l'état de charge du noeud.

D'une manière générale cet outil peut être une bonne aide pour la mise au point des algorithmes de régulation dynamique de charge qui possèdent un élément de contrôle centralisé. Il reste cependant spécifique de la programmation logique.

2.1.7 D'autres évaluateurs

Les évaluateurs de performances sont nombreux et les façons de réaliser les évaluations varient aussi considérablement. On trouve par exemple le travail présenté par C. Jacqmot et E. Milgrom qui fonctionne par prises de mesures pour étudier les paramètres des algorithmes d'équilibrage de charge et leurs répercutions sur les performances [JM96], [JM93]. Ils proposent une modélisation de la performance du système par un polynôme dont les termes et les coefficients représentent les paramètres de l'algorithme de placement à étudier. L'idée de base est d'avoir une étude complète de l'algorithme d'équilibrage en utilisant cette modélisation pour pouvoir prédire les performances pour différentes valeurs des paramètres en faisant seulement un nombre minimal de test.
D'autres évaluateurs permettent l'étude des performance des systèmes à mémoire partagée (Tango Lite [Her93]) ou bien se focalisent sur la proposition de langages de prototypage commun dans lesquels les applications parallèles peuvent être développées indépendamment de la machine cible (PROTEUS [gr:94]). Le simulateur RPPT [CDJ+91] [Jum93b] [Jum93a] permet aussi d'étudier les algorithmes de placement dynamique de processus mais sur des machines de petite taille car les noeuds et leurs configurations sont décrits un à un par l'utilisateur. Nous trouvons d'autres simulateurs où le placement de l'application est statique et doit être spécifié dans un fichier de configuration avant la simulation (PARSIM [SFWS92]).

Comme nous l'avons aperçu, il existe un grand nombre d'outils pour aider à l'amélioration des performances des systèmes parallèles. Dans cette section nous avons présenté brièvement quelques travaux qui s'intéressent à la mise au point et à l'étude des performances des applications en fonction des stratégies de placement. Plusieurs travaux considèrent certaines caractéristiques des programmes pour décider du placement des processus, soit de façon statique soit dynamiquement. Deux types d'applications sont analysées : des programmes concrets écrits dans un langage de programmation (programmes scientifiques en C, Fortran, Prolog par exemple) ou bien des programmes qui ont pour seul but de créer des charges (sur les processeurs ou sur les communications) sans donner des résultats précis. Pour la plus grande partie des travaux les mesures de performances se font sur un système physique composé de plusieurs processeurs.

Dans la section suivante nous allons présenter un outil construit pour permettre la mise au point et la comparaison d'algorithmes d'allocation dynamique. Il s'agit du simulateur SIMAD (SImulateur de Méthodes d'Allocation Dynamique) qui, comme beaucoup d'outils présentés, est constitué de 3 parties principales : la définition d'un programme, une machine et une méthode d'allocation dynamique. SIMAD, c'est un simulateur séquentiel qui peut être exécuté sur une seule station de travail. Le type d'application utilisée est de type synthétique et la décision du placement des processus ne dépend que de l'état de charge du système (partiel ou global).

2.2 Le simulateur SIMAD : présentation

SIMAD est la plateforme de simulation que nous avons utilisée et améliorée dans le cadre de notre travail. Ce chapitre est organisé de la façon suivante : premièrement nous allons montrer comment est formée une configuration de simulation qui représente la structure générale de SIMAD, ensuite nous décrirons l'implémentation de chacun de ses composants. Dans la deuxième partie de ce chapitre nous présenterons les différents résultats fournis par SIMAD.

2.2.1 Présentation générale

L'objectif principal de SIMAD est d'offrir une souplesse dans l'étude des méthodes de placement dynamique. En simplifiant la conception des machines et en utilisant une description générale du comportement des programmes parallèles, SIMAD permet d'évaluer les algorithmes de placement dynamique dans des contextes généraux en utilisant différentes machines et différents types de programmes, ou bien dans un contexte particulier en utilisant une machine destinée à exécuter une application précise.

Le simulateur SIMAD a été développé en 1994 par Gilles Pavard au sein du laboratoire HEUDIASYC, de l'Université de Technologie de Compiègne. Cet outil a été l'objet d'étude d'un projet de DEA qui fait une comparaison de SIMAD avec d'autres simulateurs du même genre [Yah94].

SIMAD est un simulateur séquentiel (un seul processeur est nécessaire pour son exécution) qui possède un fonctionnement basé sur l'occurrence de certains événements (simulation à événements discrets [Ler80]). Cet outil contrôle les interactions de trois parties principales (figure 2.1) : le mécanisme qui définit une machine parallèle (MIMD à mémoire distribuée), le mécanisme qui décrit une application avec des processus qui n'ont pas de mémoire commune et le mécanisme qui rend possible le placement dynamique des processus sur les processeurs.

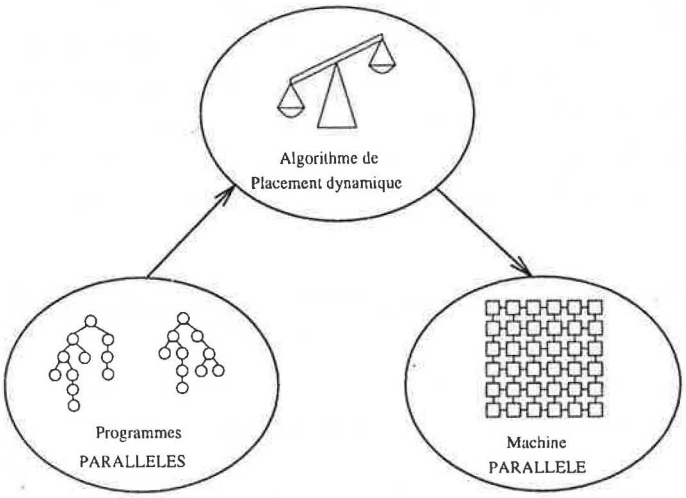


FIG. 2.1 - *Configuration de simulation pour SIMAD*

SIMAD est un outil ouvert qui permet l'étude et l'évolution de chacune des parties qui le composent (machine, programmes et algorithme de placement). En effet, une partie de notre travail a porté sur l'amélioration des types de modèles de programmes exécutés par SIMAD en considérant des programmes avec des graphes de communication généraux.

Après chaque simulation, SIMAD fournit un ensemble de résultats qui sont de deux natures. Un bilan numérique du fonctionnement du système est obtenu à la fin de la simulation (nombre maximum de processus exécutés, quantité de calculs effectué sur chaque noeud, temps utilisé par l'algorithme d'allocation sur chaque noeud, etc) et, pour permettre la mise au point de l'algorithme de placement dynamique, des informations graphiques et des textes sont fournis en cours de simulation (nombre de noeuds utilisés, nombre de messages sur la machine, nombre de processus sur la machine etc).

La notion de base dans SIMAD est celle de configuration de simulation. Les éléments d'une configuration forment la structure générale de SIMAD ce sont la définition d'une machine parallèle, la description d'un ou plusieurs programmes parallèles et une méthode de placement dynamique qui allouera les processus du programme sur les processeurs de la machine.

Etant donnée une configuration, SIMAD simulera l'exécution de l'ensemble des programmes sur la machine parallèle en utilisant l'algorithme de placement à étudier. Dans les points suivants nous allons montrer comment sont définis les composants d'une machine parallèle. Ensuite nous expliquerons comment les programmes sont construits et quel types d'instructions ils effectuent. Finalement nous parlerons de la façon d'implémenter un algorithme de placement dynamique.

2.2.2 Définition d'une machine dans SIMAD

Une machine qui fait partie d'une configuration de simulation dans SIMAD, est modélisée par un graphe orienté connexe de noeuds. Les noeuds sont décrits par une mémoire locale, une unité de traitement, et un mécanisme pour traiter les communications entre les noeuds à travers des liens de communications unidirectionnelles.

Les composants d'un noeud sont montrés sur la figure 2.2, la mémoire locale est seulement associée à l'unité de traitement (processeur) pour exécuter les processus des programmes et de l'algorithme de placement dynamique. Le processeur de communications n'est pas limité en capacité de stockage. Pour simplifier le fonctionnement de la machine parallèle, SIMAD considère que les listes de messages ne sont pas limitées en volume mémoire, car cela permet de détecter lors de la simulation les problèmes provenant de la gestion des files de communication. Le noeud est capable de traiter les communications provoquées par les processus des programmes et de l'algorithme de placement, en exploitant les liens de communication.

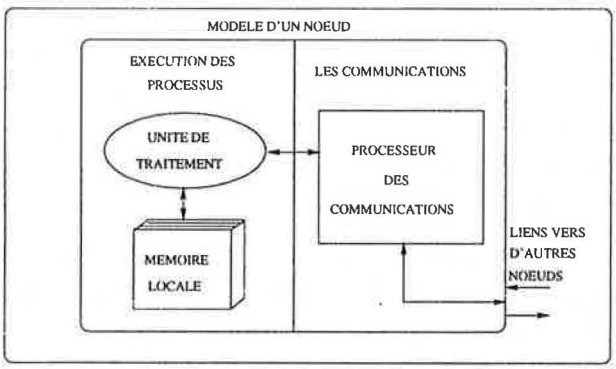


FIG. 2.2 - *Modèle des noeuds de la machine*

La partie concernant les communications gère les messages qui arrivent et qui partent du noeud. Le processeur de communications réalise toutes les opérations nécessaires

à l'acheminement des messages (routage). La partie communications est indépendante de la partie exécution des processus, à chaque instant, une instruction peut donc être exécutée en même temps qu'un message peut être envoyé ou être reçu.

Chaque noeud de la machine possède des capacités particulières. Les informations qui décrivent les caractéristiques des noeuds associent une valeur à chaque capacité, afin d'avoir une notion quantitative de la puissance du système. SIMAD prend en compte les capacités des noeuds suivantes :

- Capacité (ou taille) de mémoire
- Capacité de traitement du processeur
- Capacité de transmission de messages (ou débit des liens)
- Capacité de routage

La capacité de mémoire indique la quantité de mémoire locale associée à chaque noeud, elle est mesurée en unités de mémoire "um". Ce paramètre limite indirectement le nombre de processus placés sur un noeud, en fonction de la quantité de mémoire occupé par chaque processus.

La capacité de traitement indique la vitesse avec laquelle le processeur exécute les instructions des processus, son unité est le nombre d'instructions par unité de temps "inst/ut".

Les liens possèdent un débit qui détermine leur capacité de transmission et son unité de mesure est le nombre d'unités de mémoire transmises par unité de temps ("um/ut"). La durée de transmission d'un message est variable, c'est une combinaison entre le débit des liens et le volume du message à transmettre.

La capacité de routage est plutôt associée au type de topologie utilisée. SIMAD supporte actuellement deux types de routage pour l'ensemble des topologies. Si la topologie est une grille, un routage spécial est utilisé; ce routage exploite le numéro associé à chaque noeud (le numéro reflète la position du noeud dans la grille) pour déterminer le routage des messages. Dans d'autres cas (si la topologie n'est pas une grille), on utilise une méthode générale qui construit une table de routage pour chaque noeud de la machine qui sera ensuite utilisée pour le routage des messages. On peut voir sur la figure 2.3 un exemple de grille de taille 3x3 dont les noeuds sont homogènes.

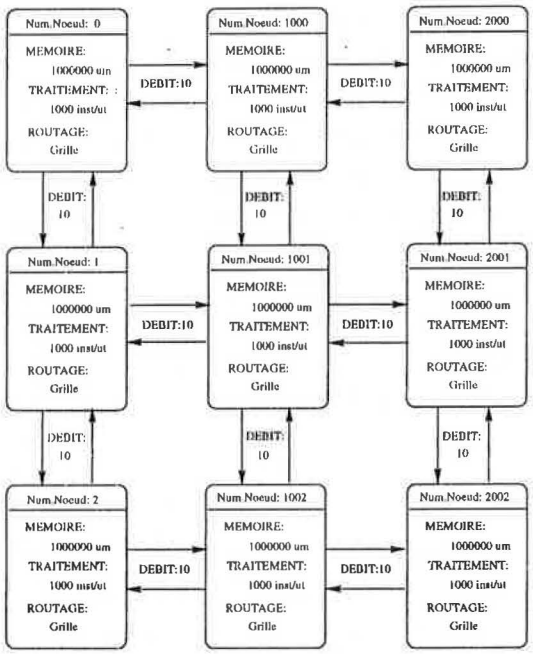


FIG. 2.3 - *Exemple d'une machine homogène (grille 3x3)*

Un outil complémentaire de SIMAD est un programme (constructeur de machines) qui génère automatiquement un fichier contenant la description d'une machine comme celle de la figure 2.3. Ce fichier est ensuite passé à SIMAD comme un des paramètres de simulation. Le fichier contient sous un certain format, les valeurs qui décrivent les caractéristiques des noeuds (débit des liens, puissance de calcul, capacité de mémoire et type de topologie.).

2.2.3 Définition d'un programme à simuler

Un programme est modélisé dans SIMAD par un ensemble de processus asynchrones qui n'ont pas de mémoire commune et qui communiquent donc entre eux par l'envoi et la réception de messages. Dans ce modèle, les processus sont vus comme des entités qui utilisent les ressources de la machine. A cet effet, SIMAD prend en compte les ressources concernant le calcul, les communications et la mémoire.

Afin d'étudier les performances d'un programme exécuté sur différentes machines et avec plusieurs algorithmes de placement, les programmes sont décrits de façon déterministe (toujours la même trace d'exécution). De cette manière un même programme contiendra des processus qui font toujours les mêmes quantités de calculs et de communications (avec les mêmes graphes de création et de communication).

La figure 2.4 illustre un exemple de programme en SIMAD constitué de 11 processus suivant des contraintes de précédence (ordre de création) et communiquant entre eux. Dans le modèle, le graphe de créations des processus est toujours un arbre. Initialement le graphe de communications était égal au graphe de créations, mais une amélioration que nous avons apportée permet d'avoir des programmes avec des graphes de communications généraux comme l'exemple montré sur la figure 2.4.

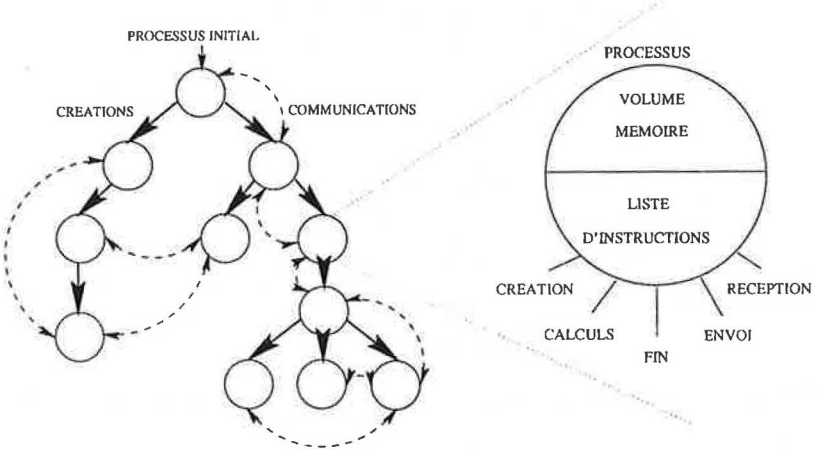


FIG. 2.4 - *Modèle de programme dans SIMAD*

Un processus est décrit par un identificateur numérique (entier unique ≥ 0), par un volume mémoire et par la liste d'instructions qui s'exécuteront séquentiellement sur le simulateur. Les instructions des processus modélisées dans SIMAD sont :

- L'instruction de création. Cette instruction détermine l'ordre et les conditions d'apparition d'un processus sur la machine.
- L'instruction de traitement. Avec cette instruction, le processus utilise le processeur pendant un certain temps (réalisation de calculs).
- L'instruction de réception d'un message. Le processus reste bloqué dans l'attente d'un message provenant d'un autre processus. Si le message est présent l'instruction de réception s'achève et le temps d'exécution est pris en compte. Dans le cas contraire le processus est rendu inactif.
- L'instruction d'envoi d'un message. Un processus peut envoyer un message sans rester bloqué.
- L'instruction fin. Cette instruction marque la terminaison d'un processus (et implique une libération d'espace mémoire).

Sur la figure 2.5 est représenté un exemple détaillé de programme. Tous les processus (du 1 au 5) sont issus du processus initial 0. D'abord les processus 1 et 2 sont créés par le processus 0, ensuite les processus 3 et 4 par le processus 1 et le processus 5 par le 2. Les processus 3, 4 et 5 ne créent pas d'autres processus, ils font des calculs (utilisent le processeur pendant un temps donné) et communiquent avec leurs processus pères pour leur envoyer des résultats. Pour le type de programme,

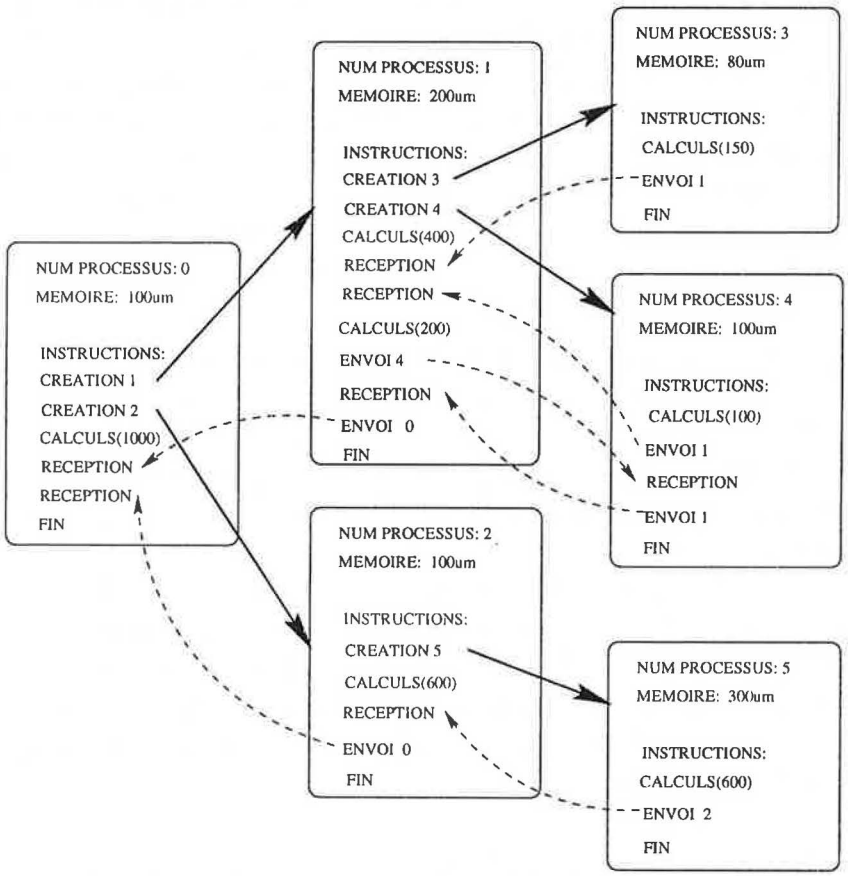


FIG. 2.5 - *Exemple de programme dans SIMAD*

exécuté dynamiquement, montré sur la figure 2.5, les instructions des processus ont besoin de quelques paramètres spécifiques. Le tableau suivant montre les paramètres

associés à chaque instruction.

Instruction	Paramètres
Création	numéro du processus à créer
Calcul	Volume de calculs à réaliser
Envoi	Volume de données à envoyer numéro du processus destinataire du message
Réception	
Fin	

Pour ce type de programme les communications se font entre père et fils, les réceptions ignorent les processus expéditeurs des messages et les messages ne contiennent que des blocs de données. SIMAD reçoit comme donnée d'entrée un fichier sous un format spécial contenant la définition d'un programme. Il existe un outil complémentaire de SIMAD qui se charge de construire de façon automatique ces fichiers. Avec cet outil générateur de fichiers on peut étudier un grand nombre de programmes différents en faisant varier, par exemple, le nombre total de processus, le nombre de créations par processus et les besoins en calcul et en communications.

Ce type d'application (communications père-fils) est très général, on peut étudier son exécution sur différentes machines, avec différents algorithmes de placement dynamique. Cependant d'autres types d'applications construites à partir des primitives provenant de la programmation parallèle [Ban91] (qui permet la construction de programmes avec un graphe de communications général et l'utilisation de différents types de messages) seraient d'un grand intérêt d'étude. Une partie du travail de cette thèse a été consacrée à la définition d'un langage de haut niveau qui permet à l'utilisateur d'extraire le comportement de ses propres programmes pour les étudier avec SIMAD. Le langage a été défini en respectant toujours l'idée de base qui est l'abstraction du comportement des processus vis à vis de l'occupation des ressources (processeur, mémoire et communications). Nous présenterons dans le chapitre 3 cette nouvelle contribution à SIMAD.

2.2.4 Définition d'un algorithme de placement dynamique pour SIMAD

La définition d'un algorithme de placement dynamique (APD) est la troisième composante d'une configuration de simulation dans SIMAD. L'APD peut faire partie d'un système d'exploitation d'une machine parallèle [EM94, EKMT94] et son objectif est la prise de décisions concernant l'exécution locale des processus ou bien leur placement sur d'autres noeuds de la machine.

SIMAD laisse aussi la gestion des processus d'un même noeud (exécution concurrente) à la charge de l'algorithme de placement dynamique. L'utilisateur a donc l'option de modifier la partie associée à l'ordre d'activement des processus sur un même noeud (en fonction de priorités, de façon cyclique, etc) ou bien de s'intéresser seulement au placement des processus.

L'algorithme de placement dynamique est présent sur chacun des noeuds de la machine et il peut être formé par plusieurs processus ou modules qui accomplissent des tâches spécifiques, par exemple le processus chargé de l'envoi d'information ou bien le processus chargé du traitement des messages arrivés contenant de l'information utile pour le placement. SIMAD offre la possibilité de définir des modules de APD avec une fonctionnalité différente pour chacun des noeuds, avec l'intérêt de pouvoir ainsi représenter des algorithmes hiérarchiques, centralisés, distribués ou hybrides. L'algorithme de placement devient dans ce cas variable en fonction du numéro de noeud sur lequel il est exécuté, car sa description complète reçoit comme paramètre le noeud et le motif de l'appel. La figure 2.6 illustre le modèle général de l'algorithme de placement dynamique dans SIMAD. L'APD réagit lors d'un appel normal (ordonnancement) et lors d'un appel pour création d'un processus (placement).

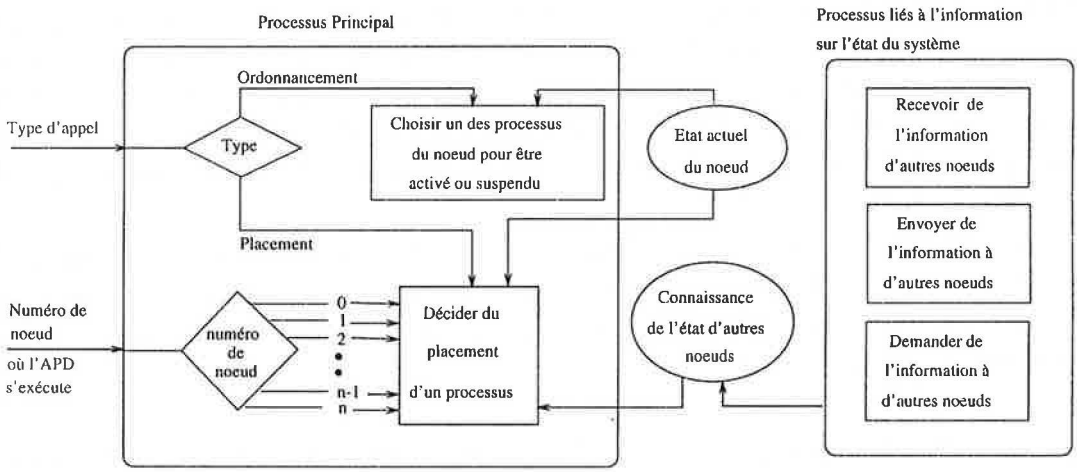


FIG. 2.6 - *Algorithme de placement dynamique dans SIMAD*

Si l'appel est de type ordonnancement, l'APD doit déterminer le prochain processus devant être exécuté sur le noeud. L'ordonnancement des processus a actuellement deux modes de fonctionnement possibles. Dans le premier mode le processeur est partagé équitablement par tous les processus, tandis que dans le deuxième mode on favorise les processus qui pourraient débloquer d'autres processus. Dans les deux cas, les processus sont exécutés pour une durée maximale TEMPS_PAR_TACHE. Ce temps détermine la fréquence d'appel de l'APD lorsque le processus activé fait

des calculs (utilise le processeur).

Si le type d'appel à l'allocateur est de type placement, l'APD doit prendre en charge le placement du processus considéré. La décision de placer un processus est prise au moment de création du processus, et lorsqu'un processus a été alloué sur un processeur, il ne sera pas déplacé pendant son exécution (pas de migration).

2.2.4.1 Les informations disponibles pour l'algorithme de placement

Dans SIMAD, chaque module de l'APD peut disposer d'informations concernant l'état de charge du noeud qu'il occupe et les processus qui s'y exécutent. Les informations directement disponibles pour l'APD concernent la mémoire, l'utilisation du processeur et les communications. Ces sont des informations ponctuelles sur l'état courant d'un noeud dans un instant donné. Si l'on souhaite disposer d'informations sur une période de temps plus longue (historique des statistiques d'exécution), l'APD peut les gérer dans chacun des noeuds.

SIMAD laisse à la charge de l'APD la tâche de gérer le volume de mémoire libre du noeud en lui permettant de réserver l'espace pour les processus qui ne sont pas encore sur le noeud. Un APD peut connaître à tout instant le volume total de mémoire, de mémoire occupé et de mémoire libre du noeud sur lequel il s'exécute. Ces informations pourraient être utiles dans l'estimation de l'état de charge du noeud.

Par rapport à l'utilisation du processeur, deux listes sont présentes sur chacun des noeuds. Ces listes regroupent d'une part les processus exécutables et d'autre part les processus en attente de communication. La longueur des listes peut être utilisée comme un indicatif de l'état de charge du noeud.

Pour les communications, on dispose du nombre de messages en attente de communication sur le noeud ainsi que de leur volume. On peut connaître aussi la répartition des messages en consultant chacun des liens de la machine. Une mesure de la charge du noeud pourrait être le nombre de messages en attente sur le noeud, en retirant les petits messages de placement qui peuvent être négligés.

Les informations concernant les processus des programmes sont obtenues au moment de leur création. A cet instant l'APD dispose uniquement de leur volume mémoire (à présent le volume mémoire d'un processus ne varie pas dans l'exécution). En cours de simulation, l'APD dispose de la date de début d'exécution du processus et de sa dernière date d'exécution (date à laquelle la simulation de son exécution a été prise

en compte par le simulateur) ainsi que de l'état du processus (bloqué en attente de communications ou actif s'il est en cours d'exécution).

L'APD d'un noeud peut également disposer d'informations sur les autres noeuds de la machine. Il suffit qu'il se mette en contact avec les autres APD pour réaliser des échanges de messages et avoir une plus large connaissance de l'état du système. Ces informations permettront à l'APD de prévoir quels sont les noeuds susceptibles d'accueillir des processus.

2.2.4.2 les perturbations engendrées par l'APD

SIMAD mesure les perturbations engendrées par les APD aussi bien au niveau des processeurs que des communications. Chaque fois que l'APD est appelé il renvoie, en plus des informations du placement, le nombre d'instructions qu'il a effectué. De cette manière SIMAD peut mesurer les perturbations de l'APD sur le processeur. Quant aux perturbations de communication, tout échange d'information entre plusieurs APD fait l'objet d'un message qui va être comptabilisé pour connaître le nombre des messages générés par l'APD (envoi d'information sur l'état de charge des noeuds, envoi de requêtes de placement des processus, envoi de refus/acceptation de processus, etc).

En général un APD peut recevoir 3 types de messages, soit des réponses à des demandes de placement déjà formulées, soit des demandes d'autres APD, soit des messages contenant de l'information. Lorsqu'une instruction de création de processus est exécutée, l'APD d'un noeud X peut décider de placer le processus sur le noeud courant, ou bien sur un autre noeud. Si le processus est placé localement, on réserve l'espace mémoire nécessaire au processus. Si l'APD décide de placer le processus sur un autre noeud Y il envoie un message au noeud Y contenant une demande de placement, puis poursuit son exécution sans attendre la réponse qui viendra plus tard. Cela est le fonctionnement d'une politique d'acceptation/refus des processus. Dans ce cas le message transmis contient l'identificateur du noeud de départ et celui du noeud destinataire qui permettent l'acheminement du message et de sa réponse. Le message contient aussi l'identificateur du processus à créer et sa taille mémoire. Le type du message indique une demande de placement.

Si l'on ne s'intéresse pas à l'utilisation d'une politique d'acceptation/refus, l'APD peut directement envoyer au noeud Y (dans un message) le processus à créer. Puis l'APD du noeud Y peut décider d'exécuter le processus ou bien de le retransmettre à un autre noeud. Dans ce cas, le volume du message transmis augmenterait suivant le volume du processus. L'algorithme de placement du Gradient [LK87] est un

exemple d'algorithme avec un tel comportement.

2.3 Le simulateur SIMAD : fonctionnement

SIMAD utilise des événements discrets pour faire avancer la simulation. Un événement discret est une opération élémentaire du système qui représente l'utilisation d'une ressource pendant un temps donné. Les instructions qui décrivent le comportement d'un programme et les fonctions pour implémenter un algorithme de placement dynamique, produisent des événements différents.

Les événements d'exécution sont générés pendant l'exécution des programmes par toutes les opérations qui utilisent les processeurs de la machine. Tous les messages générés par les processus des programmes ou par les processus de l'algorithme de placement produisent des événements de communication.

2.3.1 Les événements de communication

Les événements de communication regroupent toutes les opérations concernant l'utilisation du processeur de communication ainsi que les liens de communication. Les messages, dans la machine, sont créés pendant la simulation (par les instructions des programmes ou par l'APD). Dans SIMAD les principales informations caractéristiques d'un message sont :

- le volume
- le type
- la priorité
- la destination

Pour SIMAD un message est un volume de données qui a une destination (un noeud ou un processus). Dans SIMAD le déroulement des programmes est accéléré en permettant des créations non bloquantes de processus asynchrones. Cela implique qu'un processus X qui crée un processus Y, ne reste pas bloqué jusqu'à ce que le processus Y soit complètement créé. Dans SIMAD il est nécessaire de connaître la localisation physique des processus avant de leur envoyer des messages; cette information n'est disponible que lorsque les processus destinataires sont totalement créés. Pour l'envoi de messages il est donc nécessaire que le processus expéditeur s'assure que la création du processus destinataire du message a été faite. Pour résoudre ce problème,

les processus expéditeurs peuvent recevoir un message provenant des processus dont la création n'est pas certaine afin de vérifier qu'ils ont débuté leur exécution.

Dans la simulation, les messages à faible priorité (généralement contiennent des données des processus) peuvent être interrompus par des messages à une grande priorité (généralement contiennent des informations pour l'algorithme de placement dynamique). Lorsqu'un message à petite priorité est interrompu, il sera coupé en deux messages, un qui a été déjà transmis par le lien et le deuxième qui reste à transmettre. Le message déjà transmis prend le type INTERROMPU et le message qui reste à transmettre conserve son type initial.

Le type d'un message permet de différencier les messages destinés à des processus d'un programme des messages destinés à l'APD. La priorité des messages sert à favoriser l'acheminement des petits messages très utiles pour l'APD. Un message à grande priorité interrompt les messages à priorité inférieure.

L'acheminement des messages sur les liens de la machine se fait en deux phases : le traitement des messages à envoyer et le traitement des messages arrivés sur chaque lien. SIMAD associe deux listes à chaque noeud de la machine, une pour mettre les messages qui doivent être envoyés et une autre pour mettre les messages arrivés.

- Le traitement des messages à envoyer

  Cette phase est divisée en deux étapes, la première se charge d'affecter chaque message (devant être envoyé) à un lien de communication et la deuxième se charge de le transmettre sur ce lien. Lorsqu'un message doit être affecté à un lien, le simulateur utilise la fonction de routage pour déterminer par quel lien l'acheminement du message va commencer où continuer, même si le lien a déjà d'autres messages en attente.
  SIMAD effectue ce choix à cet instant pour simplifier les communications. Cela n'est pas gênant dans la mesure où les méthodes de routage utilisées ne concernent qu'un lien à la fois. Lorsqu'il est possible de choisir entre plusieurs liens, le lien le moins saturé est choisi afin d'équilibrer les communications entre les liens.
  Pour la deuxième étape, la transmission des messages, deux choses doivent être vérifiées avant d'effectuer le transfert. La première concerne la date de départ du message. Un message possède une date de départ calculé par SIMAD, le message sera transmis si la date de départ est inférieure ou égale à la date actuelle de simulation. Dans la section 2.3.3 on expliquera plus en détail les dates utilisées dans SIMAD. La deuxième vérification concerne le transfert des

messages, il s'agit de vérifier si le lien est libre ou non. Si le lien est libre le message à envoyer commence à être transmis sinon, on vérifie si le message à envoyer est plus prioritaire que le message qui occupe actuellement le lien.

- Le traitement des messages arrivés

  Dans cette phase, les messages reçus sont séparés dans deux groupes. Un groupe contient les messages qui sont partiellement arrivés et qui sont à destination d'un autre noeud. Dans ce cas, les messages sont mis dans la liste de messages à envoyer du noeud. Afin de représenter le temps nécessaire mis par le processeur de communications pour effectuer cette opération, SIMAD retarde le départ du message en utilisant une pénalisation de passage du message.
  Le deuxième groupe contient les messages reçus qui ont été totalement transmis et qui sont à destination du noeud considéré. C'est dans cette partie que les messages sont différenciés par leur type et ils sont ajoutés soit dans une liste spécifique à un processus soit, dans la liste des messages de l'APD.
  SIMAD ignore les messages d'arrivée de type INTERROMPU et ne les prend en compte que lorsque la totalité du volume des messages est acheminée du noeud de départ au noeud d'arrivé. L'objectif du traitement des messages dans SIMAD est de représenter les pénalisations engendrées par les communications et non de modéliser de manière précise la reconstitution des messages par le noeud avec tous les morceaux interrompus.

2.3.2 Les événements d'exécution

Lorsque les processus du programme ou de l'APD ont besoin d'utiliser le processeur, le module de traitement de calculs est appelé. Dans une première étape ce module appelle l'APD pour choisir le processus qui devra être exécuté et le temps processeur qui lui est alloué.
Dans une deuxième étape, le simulateur exécute une instruction du processus choisi. Les instructions des processus présentées dans la section 2.2.3 (création, envoi, réception, calcul et fin) décrivent les événements d'exécution qui influencent le fonctionnement du processeur. Si le temps alloué par l'APD pour exécuter l'instruction d'un processus n'est pas suffisant, l'instruction s'exécute partiellement. Cela ne peut arriver que dans le cas d'une instruction de calcul, SIMAD traduit cette exécution partielle par une diminution du nombre d'opérations restant à exécuter par le processeur.

Dans la machine le fonctionnement des noeuds est indépendant. Sur chaque noeud

l'utilisation du processeur est indépendante des communications, et pour les communications chaque lien fonctionne aussi à son propre rythme. La gestion de tous les événements qui peuvent survenir au même instant, s'effectue à travers une date de disponibilité pour toutes les ressources. Dans la section suivante nous montrons l'utilisation de telles dates.

2.3.3 L'utilisation de dates dans SIMAD

SIMAD utilise une simulation à événements discrets. Pour chaque événement possible il y a une fonction qui détermine les événements suivants et leurs dates d'activation. Pendant une simulation, SIMAD actualise une liste d'événements ordonnée par rapport aux dates d'activation et active l'événement suivant pour faire avancer le temps.

Les noeuds de la machine ont une date de traitement associée. SIMAD conserve une liste des noeuds ordonnée par rapport à cette date. Pour déterminer l'événement qui doit être simulé à un instant donné, le simulateur choisit d'abord le noeud dont la date de traitement est la plus petite puis choisit l'événement qui exploite une ressource à cette date. Après la prise en compte d'un événement, la date du noeud est mise à jour et la liste des noeuds est réordonnée. La simulation se termine lorsqu'il n'y a plus de processus à exécuter.

Les différentes dates utilisées dans SIMAD sont groupées en trois blocs : les dates relatives aux noeuds, les dates relatives aux processus et les dates relatives aux messages. La figure 2.7 illustre ces trois blocs en montrant plus en détail les dates qu'ils contiennent.

2.3.3.1 Les dates relatives aux noeuds

La date de traitement du noeud indique l'ordre de consultation dans la simulation et détermine l'instant où un nouvel événement doit être pris en compte sur le noeud. Chaque événement a une durée strictement positive pour indiquer le temps d'utilisation de la ressource considérée. L'événement choisi est pris en compte en appliquant ses conséquences, puis en ajoutant sa durée aux ressources qu'il exploite, finalement il est supprimé de la liste des événements.

La date d'utilisation du processeur indique la date à partir de laquelle le processeur peut effectuer une nouvelle opération (du programme ou de l'APD). Cette date évolue avec la durée des événements qui occupent le processeur. La date d'attente

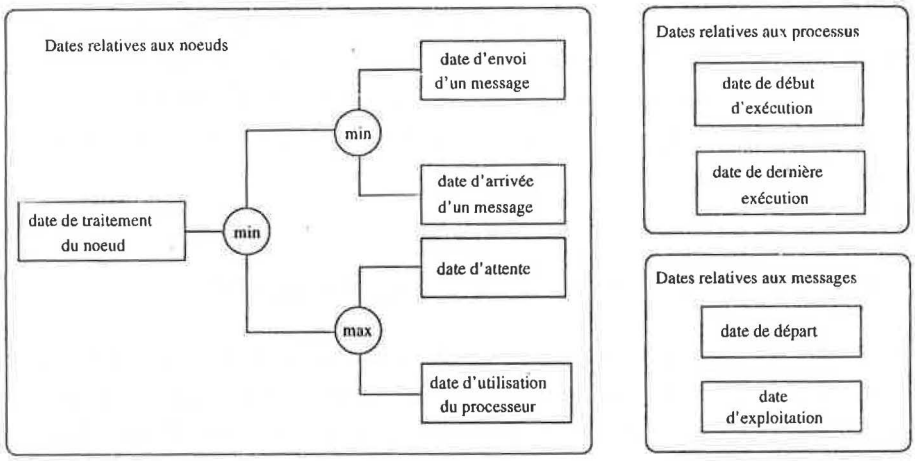


FIG. 2.7 - *Les dates dans SIMAD*

indique la date à laquelle l'algorithme de placement dynamique doit être appelé si aucun message n'est arrivé sur le noeud avant cette date. C'est le module de placement qui donne la date d'attente, pour ne pas être pris en compte elle doit être inférieure à la date d'utilisation du processeur.

La date d'envoi d'un message indique quand un message relatif au noeud, doit être envoyé. Cette date correspond à la plus petite des dates de départ des messages de la liste à envoyer, combinée à la plus petite des dates où un lien peut transmettre un nouveau message. La date d'envoi n'est pas prise en compte si sa valeur est inférieure à la date de traitement du noeud. La date de réception indique qu'un message a été reçu sur le noeud et qu'il doit être traité. Cette date correspond à la plus petite des dates de départ des messages arrivés sur le noeud, mais à destination d'un autre noeud, combinée à la date d'exploitation des messages arrivés au noeud concerné.

2.3.3.2 Les dates relatives aux processus et aux messages

A chaque processus est associée une date qui indique la dernière utilisation du processeur (date de dernière exécution) et une autre date qui indique le début de son exécution. Cette information peut servir à l'algorithme de placement lorsqu'il gère l'exécution concurrente des processus, afin de repartir équitablement le processeur entre les processus.

Les messages ont deux dates associées, la date de départ et la date d'exploitation. La date de départ indique le moment où l'en-tête du message peut être exploité. Ainsi lors de l'envoi du message sur un autre noeud, la date de départ du message

correspondra au moment où le message pourra être transmis. La date d'exploitation du message indique quand le message est totalement arrivé sur le noeud. Il peut alors être traité par une instruction de réception des processus ou par l'algorithme de placement.

2.3.4 Les durées des événements

Les durées des événements sont calculées à partir des caractéristiques physiques de la machine, des opérations des processus et de l'APD.

Pour les communications, la durée d'acheminement d'un message dépendra du débit du lien, du volume du message, des durées qui seront ajoutées par les fonctions de communication et du nombre de liens parcourus.

Pour l'utilisation des processeurs, l'événement produit par une instruction de calcul d'un processus a une durée qui est fonction du nombre d'opérations requis par l'instruction, de la capacité de calcul du processeur et du temps pris par les autres processus qui partagent le processeur avec lui (scheduler). Les instructions d'envoi, de création et de fin des processus ont une durée très courte, elles n'occupent le processeur que très peu de temps. L'instruction de réception a une durée qui dépend de l'exécution des autres processus sur le noeud. Quant aux événements générés par l'APD, les fonctions nécessaires pour implémenter l'algorithme doivent fournir une estimation du nombre d'opérations effectuées pour déduire la durée d'utilisation du processeur.

2.3.5 Les événements qui peuvent remettre en cause les dates

Dans SIMAD il y a deux événements qui peuvent remettre en cause les dates attachées à un noeud : l'envoi de messages prioritaires et la réception de messages qui peuvent rendre actifs les noeuds. Les événements restants correspondent à une exécution sur la machine qui ne peut pas être interrompue.

Lors de l'interruption d'un message par un autre plus prioritaire, tous les liens qui acheminent le message vont avoir leurs dates remises en cause. Comme il a été expliqué dans la section 2.3.1 un message interrompu est divisé en deux messages, le message qui est déjà transmis et le message qui reste à transmettre. Pour le message qui reste à transmettre, les dates de départ sont modifiées car il y a un message prioritaire qui doit être envoyé avant lui. Pour le message déjà transmis, sa date

d'exploitation est avancée car il n'a plus le volume initial du message original. Cela peut entraîner une modification des dates relatives aux noeuds et donc de leur ordre dans la liste des noeuds.

Lorsqu'un message arrive sur un noeud, il peut modifier la date d'arrivée d'un message relative au noeud si le noeud est inactif et si la date d'exploitation du message est inférieure à la date d'attente du noeud. Ainsi les messages seront pris en compte au plus tôt.

2.4 Résultats fournis par SIMAD

L'utilisation de SIMAD permet de répondre aux besoins d'implantation et d'étude d'un algorithme de placement dynamique. Il fournit une aide à la mise au point de l'algorithme et ensuite il permet de l'analyser et de le comparer avec d'autres algorithmes en utilisant plusieurs programmes et machines.

Les résultats fournis par SIMAD peuvent être obtenus de deux manières, soit pendant la simulation soit sous forme de bilan après simulation. La première option est surtout utilisée pour analyser le comportement d'un algorithme dans des cas simples avec un temps de simulation court. La deuxième option produit un bilan général de la simulation qui peut être plus longue. Ces résultats peuvent être combinés pour produire l'analyse d'un algorithme ou la comparaison de plusieurs algorithmes. Nous allons présenter ces deux types de résultats.

2.4.1 Les résultats accessibles pendant la simulation

Les résultats obtenus au cours de la simulation sont en général des résultats graphiques qui mettent à disposition de l'utilisateur un certain nombre d'informations pouvant être interprétées visuellement (figures de 2.8 à 2.14), d'autres informations sont obtenues sous forme de texte (figure 2.12).

La figure 2.8 présente la fenêtre principale d'exécution de SIMAD; dans cet exemple une machine grille de taille 4x4 est utilisée, la représentation graphique de chacun des noeuds est montrée sur la figure 2.9. Les figures 2.11 et 2.10 représentent les informations sur les programmes, pour des raisons de lisibilité elles sont surtout utiles dans le cas où le nombre de processus n'est pas trop important. Les figures 2.13 et 2.14 montrent un bilan de l'occupation de certains ressources de la machine. Ces fenêtres graphiques ne fonctionnent actuellement que pour l'utilisation de grilles

à deux dimensions. Finalement, la figure 2.12 montre quelques informations obtenues sous format de texte sur le fonctionnement du simulateur.

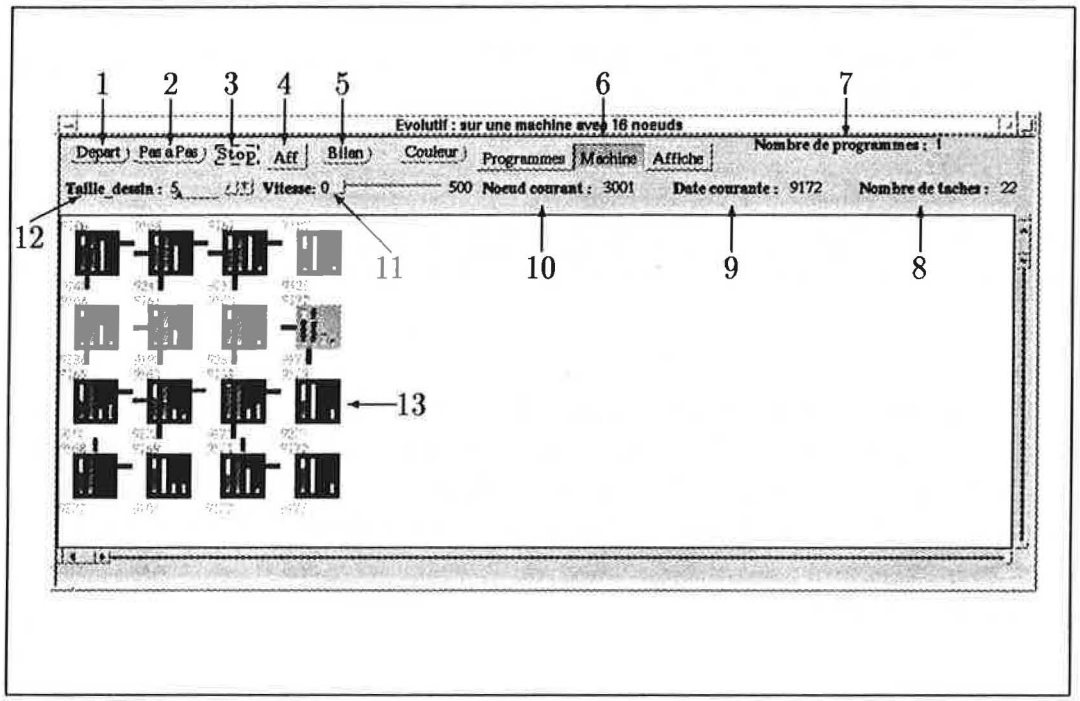


FIG. 2.8 - *Représentation graphique d'une machine dans SIMAD*

La bande supérieure de la fenêtre principale (figure 2.8) permet de choisir l'information que l'on souhaite obtenir. La signification des boutons est référencée par la liste des numéros suivants :

Numéro	Signification
1	Le bouton **Départ** sert à démarrer la simulation en continu
2	Le bouton **Pas à pas** sert à démarrer la simulation petit à petit, à chaque fois que l'utilisateur clique avec la souri sur le bouton.
3	Le bouton **Stop** arrête partiellement la simulation. Pour continuer il faut cliquer sur les boutons 1 ou 2.
4	Ce bouton sert à permettre ou non l'affichage de la machine ou des programmes.
5	Le bouton **Bilan** permet d'obtenir un bilan de ce qui s'est déroulé jusqu'à présent par rapport à l'utilisation de certains ressources (figures 2.13 et 2.14).
6	Les boutons signalés par ce numéro permettent le choix entre les différentes fenêtres d'affichage (figures 2.8, 2.11, 2.10 et 2.12

Numéro	Signification
7	Le nombre de programmes exécutées sur la machine.
8	Le nombre de processus en état d'attente de communications ou en état d'exécution sur la machine.
9	La date courante de simulation
10	Le dernier numéro de noeud simulé
11	Barre qui indique la vitesse de simulation lors d'une exécution en continu.
12	Niveau de précision d'affichage pour la machine et les programmes.

Chacun des carrés de la figure 2.8 (numéro 13) représente un noeud de la machine. Les informations graphiques de chaque noeud donnent à l'utilisateur une connaissance sur l'occupation mémoire, l'occupation du processeur, les communications et les dates de simulation du noeud. On peut voir sur la figure 2.9 un exemple de la représentation graphique d'un noeud, l'interprétation de ses composantes se trouve dans la table suivante:

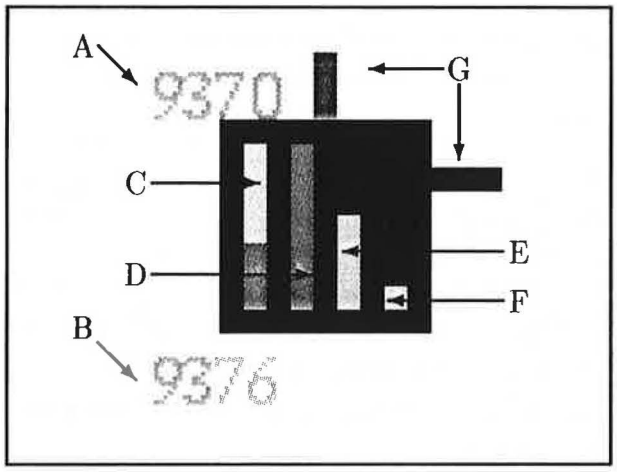


FIG. 2.9 - *Représentation graphique d'un noeud*

Numéro	Signification
A	La dernière date d'appel du noeud
B	La future date d'appel du noeud
C	La colonne de gauche indique l'occupation de la mémoire, avec 4 nuances de couleurs : mémoire libre, mémoire très occupée, mémoire réservée et mémoire normalement occupée.
D	La deuxième colonne indique le nombre de processus sur le noeud avec trois nuances de couleurs : pas de processus, processus actifs et processus en attente.
E	La troisième colonne indique le nombre de messages devant être envoyés par le noeud
F	La dernière colonne indique le nombre de messages partiellement reçus sur le noeud
G	Les liens qui sont actuellement occupés, la couleur dépend du type du message transmis.

Les informations présentes pour chaque noeud permettent de voir comment les processus sont répartis entre les noeuds et comment les capacités de communication de la machine sont exploitées. De cette manière l'utilisateur peut observer en direct le comportement du système pendant la simulation.

La figure 2.10 illustre la représentation d'un programme avec ses processus en cours de simulation. A l'aide de cette fenêtre l'utilisateur connaît l'état d'exécution de chaque programme ainsi que les processus qu'il lui reste à créer. Dans la simulation, seuls les processus en état d'attente ou actifs sont présents sur la machine. Les différentes couleurs distinguent les processus qui ne sont pas encore créés, les processus actifs, les processus en attente et ceux qui ont terminé de s'exécuter.

En utilisant le bouton 12 montré sur la figure 2.8 on obtient d'autres niveaux d'affichage. Dans la figure 2.11 on a utilisé un niveau d'affichage 6, cela a permis de voir en plus pour chaque processus sa localisation, les instructions qu'il lui reste à effectuer ainsi que ses caractéristiques. La table suivante montre la signification des lettres :

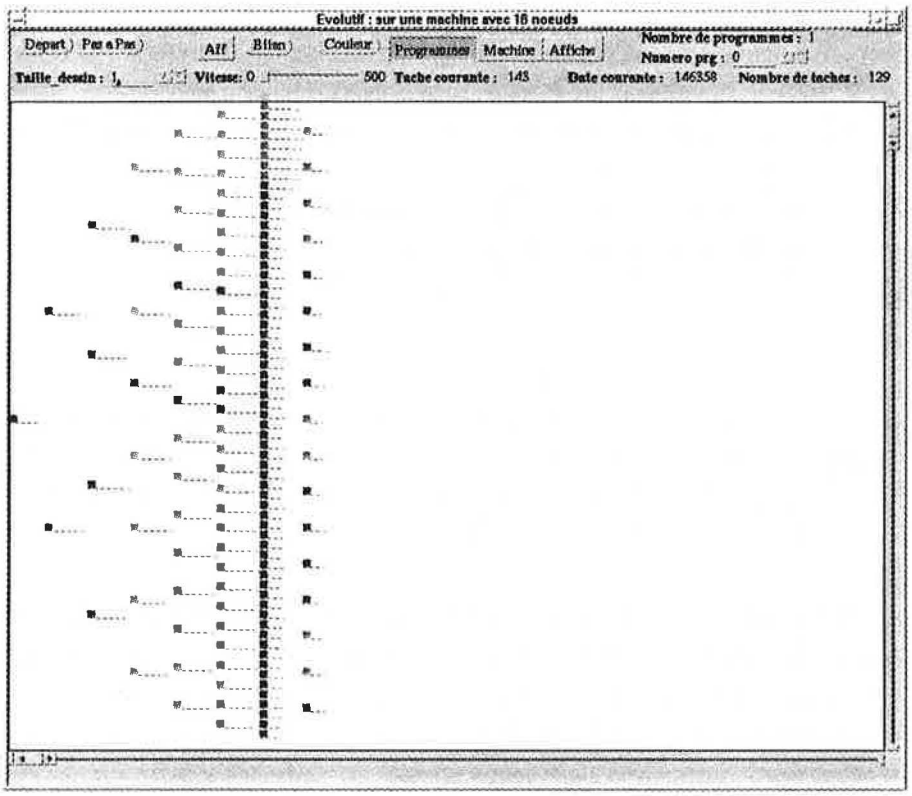


FIG. 2.10 - *Représentation d'un programme dans SIMAD*

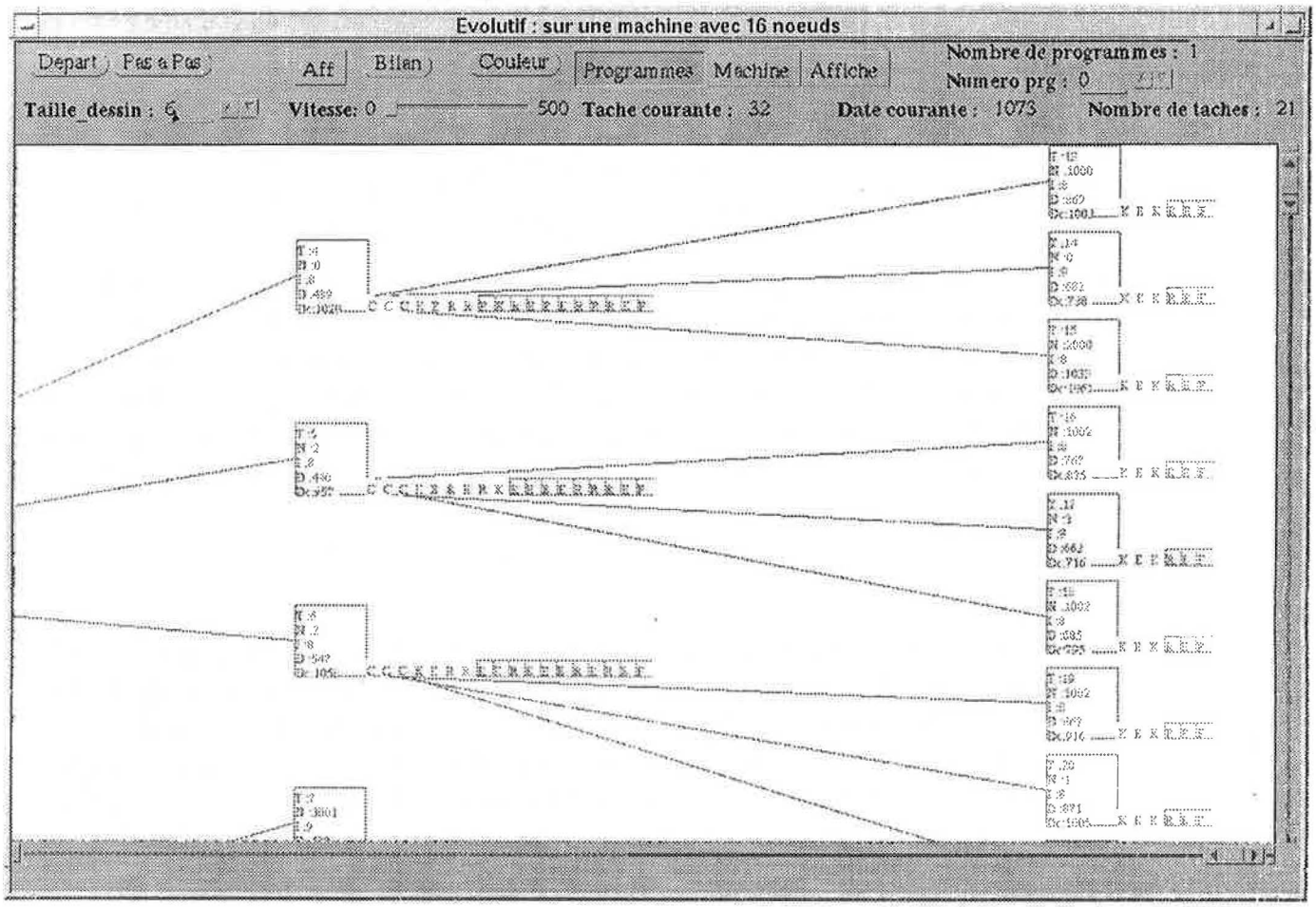


FIG. 2.11 - *Représentation d'un programme dans SIMAD*

Numéro	Signification
T	numéro du processus
N	numéro du noeud où il s'exécute
D	date de départ de l'exécution
F	date de terminaison de l'exécution
Dc	date courante d'exécution
I	instruction
M	volume mémoire
K	instruction de Calcul
E	instruction d'Envoi
C	instruction de Création
R	instruction de Réception
F	instruction de Terminaison

La figure 2.12 représente la trace partielle d'une simulation. Cette fenêtre est constituée de deux parties : la partie supérieure où l'on choisit l'élément tracé et la partie inférieure où la trace est visualisée. L'utilisateur peut ainsi tracer le déroulement de la simulation pour un ensemble choisi de noeuds et pour un critère déterminé (communications ou calculs). Les informations qui sont affichées sont des textes qui ont été prévus lors de la programmation du simulateur et de la méthode d'allocation.

Les figures 2.13 et 2.14 présentent un bilan graphique du nombre de processus exécutés par chaque processeur et de l'occupation des liens de communication. Sur la figure 2.13 les boules représentent les noeuds de la grille et leurs tailles sont proportionnelles aux nombres de processus exécutés. Sur la figure 2.14 les carrés représentent les noeuds de la grille et les boules représentent les liens de communication. La taille des boules est proportionnelle au temps d'occupation des liens de communication de chacun des noeuds. La partie supérieure des fenêtres indique l'ensemble des informations qui peuvent être visualisées. Ces informations concernent le nombre d'instructions des processus (création, calcul, réception, etc.) effectuées sur chaque noeud et les temps d'utilisation de ressources de la machine. Cette information graphique en forme de bilan peut être visualisée au cours de la simulation ou bien après simulation pour avoir un bilan final.

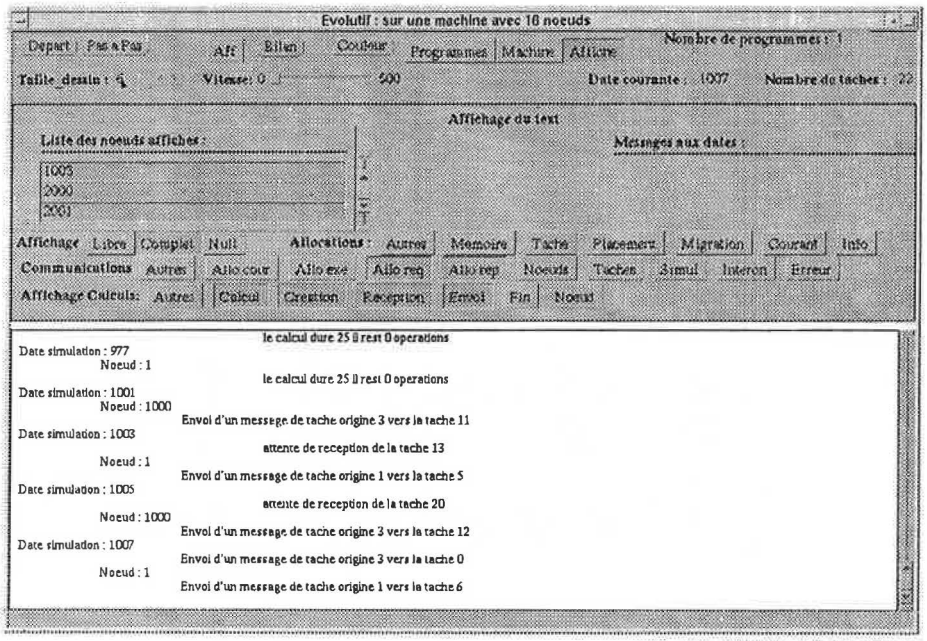


FIG. 2.12 - *Informations d'une simulation dans SIMAD*

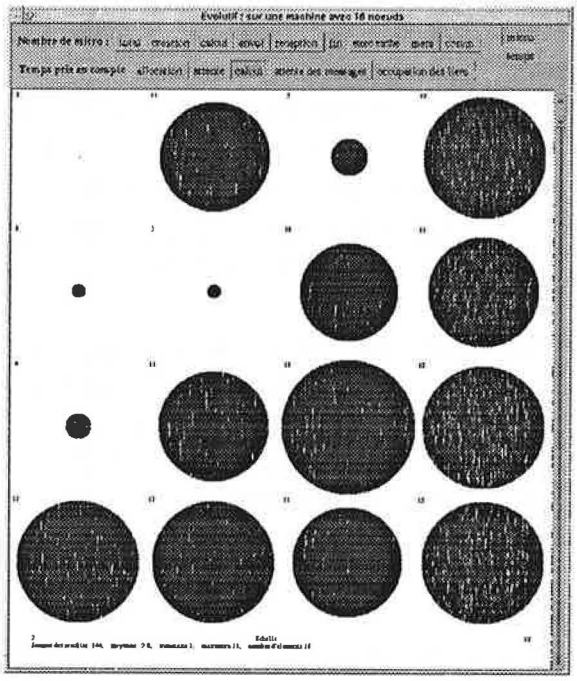


FIG. 2.13 - *Numéro de processus exécutés sur chaque noeud*

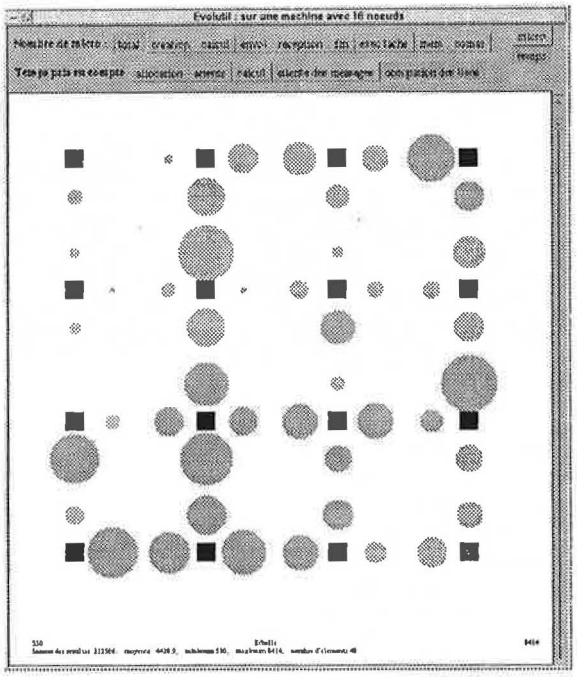


FIG. 2.14 - *Occupation des liens de communication*

2.4.2 Les résultats après simulation

SIMAD peut fournir rapidement, pour une configuration donnée, un bilan du comportement de l'algorithme de placement pour cette configuration. La simulation est plus rapide car les fonctions d'affichage sont supprimées. On peut ainsi tester des configurations avec des machines et des programmes de grande taille dans des temps raisonnables (dans nos tests approximativement de 3 secondes à 20 minutes, selon les caractéristiques des programmes et des machines).

Un jeu de test est composé de différentes configurations de simulation. Le bilan fourni pour chacune des configurations contient des informations permettant leur comparaison. Les informations concernant un jeu de test sont regroupées dans un fichier texte formé des bilans des configurations. Ce fichier texte peut ensuite être exploité en utilisant un tableur ou MATLAB.

2.4.2.1 les informations sur le programme et la machine

La description d'un programme indique le nombre de processus ainsi que des temps de référence indépendants de l'algorithme de placement utilisé. SIMAD fournit

quatre temps de référence relatifs au programme :

- Le temps parallèle correspond au temps minimal que l'on peut obtenir dans le cas où la machine est idéalement parallèle (nombre infini de noeuds tous totalement interconnectés entre eux) en distribuant un processus par noeud.
- Les temps sériels 1, 2 et 3 correspondent à l'exécution des programmes sur une machine avec un seul noeud qui n'est pas limité en mémoire. Ces trois temps se différencient par la prise en compte ou non de la mémoire et des communications dans les calculs.
  - Le temps sériel 1 considère que le volume mémoire et le volume de communications passent par un lien du réseau (on additionne le volume mémoire et le volume de communication pour les diviser par le débit d'un lien).
  - Le temps sériel 2 considère que seules les communications passent par un lien du réseau.
  - Pour le temps sériel 3 aucun volume n'est pris en compte, seuls les temps d'exécution de chacun des processus s'additionnent.

La description d'une machine indique le nombre de noeuds de la machine, son volume mémoire global et le débit moyen des liens de communication.

La première information obtenue après simulation est le temps total de simulation. Cette information est généralement celle que l'on cherche à minimiser. Les autres informations concernent l'utilisation des ressources de la machine et l'exécution des programmes.

Pour les ressources de communication, SIMAD estime le temps d'occupation minimum, maximum et moyen d'un lien ainsi que la somme de ses temps d'occupation (figure 2.14). Pour l'utilisation du processeur on possède les temps d'occupation minimum, maximum, moyen et total pour les processus de l'algorithme de placement et pour les programmes. Ces informations permettent de connaître dans quelles proportions la machine a été utilisée. Pour l'exécution des programmes, SIMAD indique combien de processus ont été exécutés et créés sur un noeud (minimum, maximum, moyenne et cumulé) comme sur la figure 2.13.

2.4.2.2 les informations sur l'évolution d'utilisation des ressources

Concernant l'évolution de l'utilisation des ressources, l'utilisateur dispose d'informations sur les communications, les processus et les noeuds. Pour les communications,

SIMAD indique la moyenne et la médiane du nombre total de messages. Les informations sur les processus concernent la quantité de processus présents sur la machine (en état d'attente ou en état d'exécution), de cette façon il est possible de savoir si le nombre de processus est important en début ou en fin de simulation. Pour les noeuds, SIMAD indique l'évolution du nombre de noeuds où il y a au moins un processus, le nombre de noeuds où il y a au moins un processus actif et le nombre de noeuds où tous les processus sont inactifs.

2.5 Remarques

Le mode d'utilisation graphique ralentit beaucoup la simulation pour effectuer les affichages. C'est pourquoi son utilisation est limitée à la mise au point ou à la compréhension du comportement d'un algorithme dans des cas bien précis. Pour expérimenter d'autres configurations plus complexes SIMAD possède le mode de fonctionnement non graphique qui fournit des résultats à la fin de la simulation.

La complexité en mémoire du simulateur est linéairement proportionnelle au nombre de noeuds et de liens de la machine, au nombre de processus du programme et au nombre de messages qui seront générés pendant la simulation. La complexité en temps de calcul est fonction du nombre d'événements générés pendant la simulation.

SIMAD est un outil d'aide au développement d'algorithmes de placements dynamique donnant des estimations de leurs performances. Cet outil nous permet d'étudier séparément les caractéristiques des programmes, des machines et des algorithmes de placement dynamique pour améliorer les performances des systèmes parallèles.

Dans cette thèse nous nous sommes intéressés à l'étude des algorithmes de placement dynamique de processus construisant des jeux de test qui permettent de comparer leurs performances. SIMAD nous a permis de réaliser nos objectifs en modifiant ses parties qui concernent la définition des algorithmes de placement dynamique et celle où l'on définit les programmes parallèles pour SIMAD.

Nous avons utilisé SIMAD pour mettre au point la définition d'un algorithme Evolutif de placement dynamique de processus présenté dans le chapitre 4. La définition d'un langage de programmation de haut niveau pour SIMAD nous a permis de créer des jeux de tests qui ont servi à tester les comportements des algorithmes de placement dynamique. Ce langage de programmation sera présenté dans le chapitre suivant.

Chapitre 3

La modélisation des programmes parallèles

Sommaire

Les programmes utilisés par les évaluateurs de performances sont principalement de deux types. Le premier type est un programme qui après avoir été exécuté fournit un résultat à un problème donné (un programme réel). Le deuxième type est un programme qui après avoir été exécuté ne fournit aucun résultat concret (programme synthétisé), cependant les ressources de la machine (en communications et en CPU) ont été utilisées comme s'il s'agissait de l'exécution d'un programme du premier type. Ce deuxième type de programme peut être représenté par un modèle, dit synthétique, de programmation [Pop90].

L'étude des algorithmes de placement dynamique de processus peut se faire : soit en utilisant de programmes parallèles réels exécutés sur des systèmes parallèles réels, soit en utilisant de programmes réels dont l'exécution est faite par simulation, soit en utilisant de programmes modélisés (à l'aide d'un langage) dont on simule le fonctionnement.

Ce chapitre commence par une description générale des programmes parallèles. Nous illustrons ensuite les différentes parties de ce chapitre par quelques programmes utilisés par des évaluateurs de performances des systèmes parallèles. Finalement nous présenterons un langage défini pour construire des programmes synthétisés dont l'exécution sera simulée par SIMAD pour tester des algorithmes de placement dynamique.

3.1 Les programmes parallèles

La programmation parallèle sous un modèle MIMD est née du besoin de séparer une quantité d'activité en plusieurs sous-activités indépendantes ou peu dépendantes. Dans le cas idéal, il suffit d'associer un processeur à chaque activité pour obtenir un gain en temps d'exécution (dans ce cas un gain linéaire : N activités indépendantes s'exécutent N fois plus vite sur N processeurs que sur un seul). Cependant, les applications réelles présentent presque toujours des dépendances. L'exécution des sous-activités est représentée par des processus qui peuvent être en relation de différentes façons :

- Indépendance entre processus
  Les processus s'exécutent en parallèle sans avoir aucune liaison dans l'exécution.

- Compétition de processus
  Lorsque plusieurs processus utilisent une même ressource l'accès à la ressource peut être contrôlée par certains mécanismes de synchronisation [BA86]. Souvent la solution est une sorte de serialisation des processus.

- Coopération de processus
  Dans ce cas il y a une participation des processus à la résolution d'un problème commun. L'échange d'informations est nécessaire.

Les langages de programmation parallèle permettent d'écrire des programmes dont les processus peuvent contenir ces trois relations. Lorsque deux processus doivent communiquer durant leur exécution, ils le font par l'envoi et la réception de messages s'ils ne partagent pas de mémoire commune. Dans [Isl94] on trouve une description des modèles de communication les plus courants.

Les langages de programmation d'une application parallèle peuvent être asynchrones (PVM, MPI [Ga94] [CC94], ASCCS [Dzi90], etc.) ou synchrones (CSP, ADA, OC-CAM, les statecharts et statemate, SCCS [Hoa78] [BCG87] [Dzi90], etc.). Dans [BST89] on peut trouver une description générale des langages de programmation pour les systèmes parallèles.

3.2 Les programmes réels

Lorsque les évaluateurs de performances utilisent des programmes réels qui calculent les résultats d'un problème, les programmes sont écrits dans un langage de programmation donné [Her93], [CDJ+91], [Chr94], [YSM95]. D'autres macro-instructions ou primitives pour la création et les communications entre processus sont disponibles par moyen de librairies. Généralement pendant la compilation du programme on ajoute du code pour calculer le temps d'exécution et pour prendre les mesures des paramètres nécessaires qui interviennent sur la performance du système.

L'outil présenté dans [SK93] étudie la performance d'un système exécutant des programmes composés par un ensemble de processus qui s'exécutent sous un ordre de précédence. Le modèle de ces programmes est un graphe dont chaque sommet est constitué d'une variable et d'une opération. L'opération d'un sommet X sera appliquée sur les valeurs provenant d'autres sommets et le résultat sera affecté à la variable du même sommet X. Une représentation graphique de cette modélisation est illustrée sur la figure 3.1.

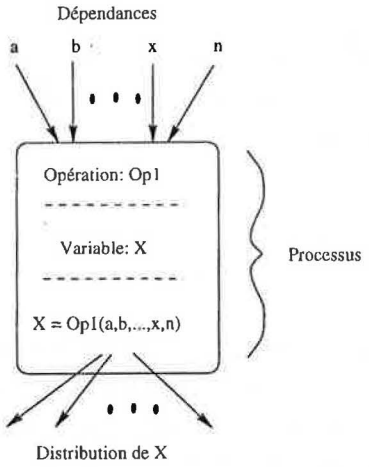


FIG. 3.1 - *Modélisation d'un processus qui possède un ordre de précédence*

Pour ce type de programmes le résultat d'un problème (arithmétique par exemple) est donné par le processus qui applique le dernier opérateur à sa variable. La performance est influencée par la synchronisation et dépendra du placement des processus sur les noeuds pour minimiser le temps d'exécution.

Le langage ATHAPASCAN-0a [Chr94], utilisé dans l'étude du découpage d'une application en sous-calculs, produit des programmes sous un modèle où chaque tâche est indépendante des autres et de sa localisation. Une tâche contient plusieurs points d'entrée (des services qui partagent le même espace mémoire) invoqués par des requêtes. Les services d'une tâche peuvent être invoqués de façon bloquante ou non bloquante par d'autres tâches. Chaque service d'une tâche peut être exécuté simultanément un nombre maximum de fois.
Les liens de communication sont seulement chargés de requêtes qui activent les services car les tâches sont indépendantes et ne font pas d'échanges d'information.

Les langages que l'on utilise pour écrire des programmes réels permettent de résoudre certains problèmes de calcul parallèle (opérations sur des vecteurs ou sur des matrices), de tri ou de pipeline par exemple. Pour obtenir une bonne implémentation, la construction de ces programmes requiert l'analyse du problème et la maîtrise du langage de programmation.
Dans la section suivante nous allons présenter le deuxième type de programme qui ne prend pas en compte les détails d'implémentation mais qui a le même effet sur l'utilisation des ressources de la machine qu'un programme réel.

3.3 Les programmes synthétisés

Lorsqu'on veut étudier et mesurer les performances d'un système parallèle vis à vis de l'exploitation des ressources et du temps d'exécution, l'utilisation de programmes qui synthétisent le comportement des programmes réels simplifie l'étude. Un programme synthétique ne donne pas de résultats relatifs à un problème concret mais génère à peu près les mêmes charges sur le système. Les charges peuvent affecter les ressources de communication aussi bien que les processeurs. Dans cette section, nous allons présenter brièvement trois exemples de programmes de ce type: les files d'attente, le langage GENESE et le langage ANDES.

3.3.1 Description des applications parallèles au moyen de files d'attente

Au lieu d'utiliser un programme parallèle qui génère un certain nombre de processus, l'introduction de lois (constante, exponentielle, générale,..) peut déterminer l'apparition de certains événements et la distribution des processus. Un exemple qui utilise ce type de modélisation est le simulateur PARSEVAL.

Le simulateur PARSEVAL [RM91] utilise des files d'attente pour représenter le comportement d'un système parallèle. Une file d'attente est définie par six paramètres A—B—C—K—m—Z. La signification est la suivante:

- A: Processus d'entrée. C'est la loi de distribution des intervalles entre les arrivées de clients (les messages) à l'entrée de la file (le processus).
- B: Processus de service. C'est la loi qui régit le service fourni par le serveur de la file. Pour A et B on peut utiliser différentes notations selon le processus, par exemple:
  - GI: loi générale, avec indépendance des variables successives
  - G: loi générale
  - L: loi exponentielle
  - D: loi constante
- C: Nombre de serveurs associés à la file
- K: Capacité maximale. C'est le nombre maximal de clients pouvant se mettre en attente dans la file
- m: Population des usagers. C'est le nombre maximal de clients pouvant passer dans la file

- Z : Mode de service. C'est la politique de gestion des clients dans la file :
  - FCFS(*First Come First Served*) : le premier client arrivé est le premier servi.
  - LCFS(*Last Come First Served*) : le dernier client arrivé est le premier servi.
  - QUANTUM : les clients sont servis à tour de rôle par tranches de temps de durée fixe.
  - PRIORITE : les clients sont classés par ordre de priorité de service.
  - PS(*Processor Sharing*) : tous les clients de la file sont en service, ils se partagent le serveur.

Ces paramètres élémentaires sont alors combinés pour former des réseaux de files d'attente, il y a donc échange de clients entre les files d'attente qui composent le réseau.

L'implantation du modèle de files d'attente peut être faite par un langage quelconque de programmation. Les charges sont générées sur les communications selon la génération des clients(messages) et sur les processeurs selon le nombre de serveurs sur chaque noeud.

Malgré la bonne description obtenue par les files d'attente des partages des ressources dans un système informatique, la modélisation est très générale. Pour décrire des comportements où la création de processus et l'envoi des messages sont plus détaillés on utilise d'autres types de programmes. Dans la suite nous donnons deux exemples de langages qui synthétisent le comportement des programmes parallèles mais qui utilisent des instructions qui ressemblent à celles utilisées dans la programmation parallèle pour la création et les communications entre processus.

3.3.2 Le langage GENESE

Pour construire des programmes sous un modèle de programme parallèle complexe nous pouvons utiliser des langages de programmation de type synthétique [Pop90]. Ces langages génèrent des programmes dont le comportement est synthétisé grâce à certaines instructions qui ont les mêmes effets sur le système qu'un ensemble de plusieurs instructions concrètes. Un programme synthétique peut aussi être utilisé pour étudier les performances d'un système parallèle. Dans la suite nous allons présenter le langage GENESE qui est basé sur les langages synthétiques pour générer des programmes parallèles généraux.

Le langage GENESE ([Hem94]) produit des programmes sous un modèle de programme qui décrit le comportement d'une application à l'aide d'un ensemble d'entités actives communicantes. La définition d'une entité (ou processus) se décompose en deux parties :

- La première partie concerne la description de l'environnement d'exécution. Cette partie donne certaines informations à l'algorithme de placement dynamique pour faciliter son travail. Les informations peuvent être de plusieurs natures :
  - Les informations obtenues dans les étapes antérieures du développement de l'application : la taille mémoire, les ressources utilisées durant l'exécution et les sites susceptibles d'accueillir l'entité.
  - Les informations obtenues à partir des simulations précédentes : par exemple les liaisons entre processus, le temps moyen pour exécuter chaque entité et le nombre moyen d'attente de messages pour chaque instance d'une entité.
  - Les contraintes d'exécution établies par l'utilisateur : les contraintes matérielles pour décrire les ressources utilisées et les contraintes de localisation pour définir le domaine de placement d'une entité.
  - Les contraintes exprimés par le programmeur : le nombre maximum d'exécutions simultanées d'instances d'une même entité et les dépendances de localité des entités.
- la deuxième partie concerne la description du comportement d'une entité. Les instructions du langage synthétique GENESE permettent de tracer les événements importants pour étudier le comportement d'une application. Plusieurs instructions ont été définies :
  - creation[(num)] nom_processus(liste_paramètres)
    Cette instruction permet la création dynamique de processus. Les paramètres nécessaires sont le site de placement **num** (paramètre optionnel), le nom du processus à créer **nom_processus** et les paramètres de création du processus **liste_paramètres**. Si le site de placement n'as pas été fixé par le programmeur, il sera choisi par l'algorithme de placement.
  - calcul[coût]
    Cette instruction permet de représenter une charge de calcul et donc une consommation du CPU en unités de temps logique. Le paramètre permet de faire varier la valeur du **coût**.
  - message[coût]processus_destinataire parm_message
    L'instruction **message** est utilisée pour envoyer un message à un pro-

cessus destinataire contenant les données dans **parm_message**. Le paramètre optionnel **coût** indique le volume du message. L'envoi d'un message n'est pas bloquant.

- attente[id_expéditeur] param_message
  L'instruction **attente** pour recevoir un message, bloque un processus jusqu'à l'arrivée du message. Le paramètre optionnel **id_expéditeur** est utilisé lorsque l'on attend un message d'un processus précis. Les données reçues sont rangées dans **param_message**. Le traitement de messages de types différents n'est pas représenté dans ce langage.
- D'autres instructions comme le branchement et les affectations sont aussi incluses. Elle permettent de décrire le comportement du processus sans générer d'événements dans la simulation.

Ce langage permet de générer une trace d'événements représentative d'une exécution réelle. L'idée étant d'améliorer les performances d'exécution en prenant en compte les informations fournies par les simulations précédentes. Les programmes en langage GENESE sont utilisés surtout dans les cas où l'utilisateur connais bien les caractéristiques des communications entre les processus. Le placement d'un tel programme dépendra des contraintes établies par l'utilisateur qui sont spécifiées dans le code du programme.

3.3.3 Le langage ANDES

Le langage ANDES est un langage synthétique développé pour être utilisé par un outil qui évalue différentes stratégies de placement statique [Kit94]. ANDES permet de décrire des programmes parallèles sous un modèle de type CSP [Hoa78] où un programme est représenté par un graphe orienté sans circuits dont les sommets sont les processus et les arcs signalent des contraintes de précédence.

Dans ce langage un processus (appelé noeud de calcul) est composé de plusieurs instructions indiquant les calculs à réaliser, les sorties et les entrées des messages. L'instruction de calcul est représentée par un coût qui peut être une constante, une distribution aléatoire ou une fonction plus complexe. Les instructions d'entrée/sortie implémentent un mécanisme de synchronisation du type logique And/Or permettant de modéliser les dépendances de données entre les processus. Comme pour le calcul, on associe aux instructions d'entrée/sortie un coût de communication lié aux caractéristiques de l'algorithme.

Dans cette première partie du chapitre nous avons présenté les différents types de programmes pouvant être utilisés par des évaluateurs de performances des systèmes

parallèles. Nous considérons qu'un langage synthétique permettant, à partir d'une description simplifiée d'une application, de générer une trace d'événements représentative d'une exécution réelle, peut être un complément important aux outils qui analysent les performances des algorithmes de placement dynamique. D'une part ce type de langage peut être assez simple et facile à mettre en oeuvre pour décrire des applications qui génèrent des processus dynamiquement. D'autre part il permet de générer des situations d'exécutions qui favorisent le test des qualités des algorithme de placement.

3.4 Un langage synthétique de programmes parallèles pour SIMAD

Avant ce travail les programmes simulés par le simulateur SIMAD (décrit dans le chapitre 2) possédaient un graphe de communications en forme d'arbre avec des arcs bidirectionels, en effet, les échanges étaient limités à une relation fils-père. Une partie de ce travail cherche à étendre les caractéristiques du simulateur du point de vue du type de programme simulé, car beaucoup d'applications parallèles réelles peuvent avoir un graphe de communication général. Par exemple, un programme où les processus destinataires et les émetteurs sont déterminés au cours de l'exécution.

Nous proposons la définition d'un langage de haut niveau qui décrit le comportement des applications parallèles et avec lequel on construira des types de programmes plus généraux pour SIMAD. Ce langage permet de générer des programmes avec des graphes de création et de communication plus complexes. Par exemple comme les programmes générés par le langage PVM [Ga94],[CC94] en respectant toujours l'idée de construire des programmes destinés à l'étude des algorithmes de placement dynamique et non à leur étude interne. Nous présentons au cours de ce chapitre le langage proposé qui est un langage synthétique (section 3.3.3) nommé LASSIMAD (LAngage Synthétique pour SIMAD) construit pour étudier les performances des systèmes parallèles dans SIMAD.

LASSIMAD permet de décrire le comportement d'un programme parallèle en se basant sur des instructions caractéristiques de la programmation parallèle [Ban91]. L'envoi non-bloquant, la réception bloquante et non-bloquante de messages, les calculs et la création de processus utilisés dans PVM, RPC, et autres, ont une correspondance avec les instructions de ce langage synthétique.

3.5 Structure générale d'un programme LASSIMAD

Un programme écrit dans LASSIMAD est composé de trois blocs. Le premier bloc commence avec la spécification des types des messages définis par l'utilisateur qui seront utilisés dans les communications entre processus (sauf les types réservés par SIMAD : **data et nktype**, section 3.7). Le programme continue ensuite avec la définition d'un processus principal **"main"** (obligatoire dans tous le programmes) qui contient le corps principal du programme et se charge de créer d'autres processus nécessaires pour l'exécution. Le troisième bloc est composé de la spécification des processus devant être créés. Ces derniers peuvent aussi créer d'autres processus dont la spécification doit faire partie du troisième bloc.

Dans la spécification d'un processus il y a comme première donnée le nom du processus suivi d'une liste de paramètres formels (sauf pour le processus principal "**main**"). Les paramètres peuvent servir à indiquer la quantité de calculs à effectuer, le nombre de communications à établir, le nombre de créations, l'identificateur associé à un processus, etc. Ensuite le corps du processus est défini, il contient la taille mémoire nécessaire pour l'exécution du processus, la définition des variables à utiliser localement et la liste d'instructions à exécuter regroupées par les symboles **{** et **}**.
Le texte suivant montre un exemple de structure d'un programme écrit dans LASSIMAD où l'utilisateur définit deux types de messages (**TypeSynch** et **TypeFin**) qui seront utilisés dans les communications des processus. Le processus **main** occupe 500 unités de mémoire (um) et utilise les variables entières **id** et **i** dans son exécution. Dans le troisième bloc il n'y a que le processus **proc1()** qui n'as pas de paramètres et occupe 1000 um, une seule variable **k** est définie. Les processus **main** et **proc1** effectuent m et n instructions respectivement (où **insti**, $1 < i <= m$ et **instj**, $1 < j <= n$ représentent des instructions dans le langage LASSIMAD).

```
LIST MESSAGES
Type1, Type2, ... TypeN

main
memory= 500;
var id,i;
{ inst1; .....; instm; }

process proc1 ()
memory= 1000;
var k;
```

```
{ inst1; .....; instn; }
    .........

process procN ()
memory= 1000;
var k;
{ inst1; .....; instn; }
```

Les modes d'exécution dans le langage sont les suivants :

- L'exécution séquentielle

  Le corps d'instructions d'un processus est construit à partir d'un ensemble d'instructions basiques exécutées par ordre d'apparition dans le texte (séquentiellement). Comme dans la programmation séquentielle, certaines instructions sont chargées de diriger le flux de contrôle (instructions conditionnelles et répétitives).

- L'exécution parallèle

  Dans le corps d'instructions d'un processus il peut être spécifié un certain nombre d'instructions pour la création de processus. Une fois que le simulateur a pris en compte une instruction de création, l'exécution du processus créateur continue avec l'instruction suivante. Le langage permet de générer l'exécution parallèle d'autant de processus qu'il y a d'instructions de création **spawn()**.

L'utilisation de variables et de structures de données est nécessaire. Pour compléter ce langage il est possible de manipuler des variables entières simples et des tableaux d'entiers qui sont indispensables pour garder les identificateurs des processus et pour les compteurs dans les instructions de contrôle du programme.

3.6 Les instructions de base

Dans cette section, nous allons décrire quelques instructions basiques du modèle et leur fonctionnement. Les primitives de communication entre processus seront expliquées plus en détail dans la section suivante.

3.6.1 La création et l'identification des processus

Pour créer un processus on utilise la fonction **spawn()**, en indiquant le nom du processus à créer et ses paramètres. La fonction retourne un identificateur entier unique (déterminé par le simulateur) associé au processus. Un exemple d'utilisation est le suivant :

```
process Proc1()
memory = 50;
var id1,id2;
    {
        id1 = spawn(procA);
        id2 = spawn(procB,300,4);
    }
```

La première instruction du processus Proc1 crée le processus **procA** qui n'a pas de paramètres, l'identificateur de ce processus sera assigné à la variable id1. La deuxième instruction crée un processus **procB** avec deux paramètres entiers. La variable id2 gardera l'identificateur unique du processus **procB**.

Lorsqu'un processus effectue une instruction de création, le processus créé commencera à s'exécuter en parallèle avec son créateur dès qu'il sera affecté à un processeur (un noeud) choisit par le module de placement. Il est possible qu'un processus **Proc1** crée un autre processus **Proc1** qui s'exécutera de manière indépendante, le seul détail à surveiller est la création indéfinie de processus Proc1.

Dans SIMAD, la création d'un processus est associée à la création d'un nouveau contexte d'exécution (variables locales et pointeurs d'instructions). Le pointeur d'instructions spécifie l'instruction à exécuter et lorsqu'il atteint la dernière instruction du processus, le simulateur libère l'environnement créé.

A la différence du langage PVM où on utilise des fonctions pour obtenir l'identificateur du processus père et l'identificateur local du processus, dans ce langage synthétique on utilise directement les valeurs de deux constantes; un processus connaît automatiquement l'identificateur du processus qui l'a créé (processus père) en utilisant la variable réservée **parent**, de la même manière il a accès à son propre identificateur par l'accès de la variable réservée **mytid**.

3.6.2 L'instruction de calcul

La manipulation des données locales ou des messages reçus par un processus est synthétisé par l'instruction de calcul **compute()**. Cette instruction permet de représenter une charge de calcul dont l'analyse n'as pas d'intérêt direct qui puisse aider dans l'exécution de l'application. L'instruction a comme paramètre une valeur entière qui indique la quantité de calculs à réaliser par le processus. Les calculs servent à décrire d'une façon générale le temps de consommation de CPU par un processus. Voici un exemple très simple qui montre le processus Proc1 dont le seul but est d'effectuer 25300 unités de calcul

```
process Proc2()
memory = 10;
    {
        compute(25300);
    }
```

cela implique l'exécution de 25300 unités de calcul qui seront effectuées pendant un temps qui dépend de la puissance de calcul d'un noeud et du nombre de processus en exécution concurrente sur le même processeur.

3.6.3 Les instructions de contrôle

Dans le langage, les instructions s'exécutent selon une des trois possibilités du flux de contrôle : la succession d'instructions (séquentiel), la conditionnelle (**if-else**) et la répétitive (**for**). Ces instructions fonctionnent comme dans le langage C. Un exemple est le suivant :

```
process Proc3(param1)
memory= 40;
var i;
    {
        if (param1 > 0)
            compute(25300);
        else
            for( i =1 ; i < 10; i=i+1)
                instj;
    }
```

Dans cet exemple, le processus **proc3** exécutera 25300 unités de calcul si le paramètre **param1** est supérieur à zéro, sinon il exécutera 10 fois l'instruction **instj** qui

représente une instruction dans le langage.

3.7 Les primitives de communication

Les processus des applications parallèles exécutées sur des machines MIMD à mémoire distribuée, communiquent par l'envoi et la réception de messages. Le langage proposé définit deux primitives de communication pour l'émission et la réception de messages.

Pour envoyer un message, on utilise l'instruction **send()** qui a comme premier paramètre l'identificateur du processus destinataire du message. Les paramètres qui suivent sont le type du message et les données à envoyer. Le type du message peut être une des possibilités suivantes :

- Le type réservé par SIMAD : **data**
  Ce type est utilisé lorsqu'on veut juste représenter une charge importante sur les communications sans avoir l'intérêt principal de connaître le contenu des données envoyés. Un exemple d'utilisation est :

  send(id,**data**,3500);

  Cette sentence implique l'envoi d'un message au processus sous l'identificateur **id**. Le message contient des données qui occupent 3500 unités de mémoire. Cette quantité sera prise en compte par le simulateur au moment de calculer la date d'arrivée du message.

- Un type de message défini par l'utilisateur
  Dans ce cas l'utilisateur peut envoyer plusieurs paramètres qui généralement contiennent des variables concernant des identificateurs d'autres processus pour établir des communications. Le volume du message transmis est formé par le numéro de variables entières à envoyer. Un exemple est l'instruction suivante où l'on envoie un message au processus **id**, le type du message est typeSynch et le contenu est une seule variable **idfrere**.

  send(id,**typeSynch**,idfrere);

Le langage ne permet pas l'envoi d'un message à un groupe de processus car l'opération de diffusion n'est pas actuellement implantée sur SIMAD. Cependant cette opération peut être traduite par plusieurs envois de messages.

Un processus peut recevoir un message en utilisant l'instruction **recv()** qui a comme premier paramètre l'identificateur du processus expéditeur du message. Les paramètres suivants correspondent au type de message attendu et aux variables que l'on s'attend à recevoir (si c'est le cas). Pour cette instruction l'identificateur du processus expéditeur peut être une des deux possibilités : soit l'identificateur spécifique à un processus (défini par l'utilisateur), soit il n'y a pas d'identificateur précis, dans ce cas on utilise la variable réservée par SIMAD : **nktid**.

Utilisant la deuxième possibilité, **nktid**, le processus pourra recevoir un message provenant de n'importe quel autre processus. Après la réception du message la valeur concernant l'identificateur du processus qui a envoyé le message sera gardé sur la même variable **nktid**.

Le paramètre concernant le type de message à recevoir a aussi deux possibilités; la première est un type de message défini par l'utilisateur et la deuxième est le type réservé par SIMAD : **nktype**. Ce dernier fonctionne comme dans le cas de la variable **nktid**, c'est-à-dire que dans la réception on peut attendre l'arrivée d'un message de type quelconque. Lorsque la réception est faite, le type arrivé est copié dans la variable **nktype**. Un exemple d'utilisation de l'instruction **recv** est le suivant :

- **recv**(id,typeSynch);
  Cette instruction spécifie la réception d'un message provenant du processus sous l'identificateur **id**, avec un type de message **typeSynch** (le type **data** peut aussi être utilisé)

- **recv**(id,**nktype**);
  On attend ici un message provenant du processus sous l'identificateur **id** mais le type du message à recevoir peut être quelconque. Dans l'instruction précédente et celle-ci il peut y avoir une liste de paramètres après le type du message

- **recv**(**nktid**,data);
  Réception d'un message provenant de n'importe quel processus qui a le type **data**

- **recv**(**nktid**,type1,a,b,c);
  Réception d'un message provenant de n'importe quel processus. Le type du message doit être **type1** et trois paramètres sont attendus

- **recv**(**nktid**,**nktype**,a,b);
  Dans ce cas on attend l'arrivée d'un message dont le destinataire et le type du message sont quelconques, mais deux paramètres sont attendus

Le schéma de communication est asynchrone, c'est-à-dire que les déroulements des processus émetteurs et récepteurs ne s'influencent pas. Si nous considérons un système de deux processus P1 et P2 où P1 envoie un message à P2, deux situations peuvent se présenter

- Le processus P2 réclame le message avant qu'il ne soit arrivé; alors il reste dans un état d'attente jusqu'à ce que le message soit disponible
- Le processus P2 réclame le message après qu'il soit complètement transmis (il est donc disponible); dans ce cas le processus peut utiliser le message sans aucune attente

Quant au processus P1, il n'est jamais interrompu à cause de l'envoi du message.

Une dernière instruction non bloquante est l'instruction **nrecv** qui se charge de vérifier l'arrivée d'un message. En utilisant la fonction **nrecv** on peut tester si un message déterminé est arrivé complètement (donc il est accessible), on obtient alors une réponse vrai ou fausse à ce test. Les paramètres nécessaires sont : l'identificateur du processus expéditeur du message et le type du message attendu. Toutes les combinaisons de ces deux paramètres utilisées dans l'instruction **recv** sont aussi permises pour l'instruction de test **nrecv**. Des exemples d'utilisation sont les suivants :

- resp = **nrecv**(id,typeSynch);
  On vérifie l'arrivée d'un message provenant du processus sous l'identificateur **id**, le type du message attendu est **typeSynch** (on peut aussi utiliser le type de message **data**). La variable resp prendra la valeur 0 ou 1 après cette vérification.
- resp = **nrecv**(id,**nktype**);
  Avec cette instruction on vérifie l'arrivée d'un message provenant du processus sous l'identificateur **id** mais le type du message attendu peut être quelconque.
- resp = **nrecv**(**nktid**,data);
  On vérifie l'arrivée d'un message provenant de n'importe quel processus, le type du message à recevoir est **data** (on peut aussi utiliser un autre type défini par l'utilisateur).
- resp = **nrecv**(**nktid**,**nktype**);
  Dans ce cas on vérifie l'arrivée d'un message dont le destinataire et le type du message sont quelconques.

3.8 La grammaire générale de LASSIMAD

Jusqu'ici les principales instructions du langage ont été présentées. De telles instructions permettent de décrire les comportements généraux des applications parallèles. Le graphe de communications est général et les charges sur l'utilisation des ressources (communications et processeur) sont toujours représentées.
La syntaxe complète du langage LASSIMAD est spécifiée par les expressions suivantes écrites dans le format BNF.

$< Program >::=$	$\{< liste\_types\_mess >\}^{0\|1}$
	$< def\_prog\_principal >$
	$< liste\_processus >$
$< def\_prog\_principal >::=$	**main**
	$< corps\_proc >$
$< corps\_proc >::=$	$< taille\_memoire >$
	$\{< def\_variables >\}^{0\|1}$
	{ $< liste\_instructions >$ **}**
$< liste\_types\_mess >::=$	**LIST MESSAGES**
	$< liste\_id\_messages >$
$< liste\_id\_messages >::=$	$\{< identificateur >\}^{+}$
$< taille\_memoire >::=$	**memory** $= < valeur >$;
$< def\_variables >::=$	**var** $< liste\_variables >$;
$< liste\_variables >::=$	$\{< ident\_variable >\}^{+}$
$< ident\_variable >::=$	$< tableau > \ \| \ < identificateur >$
$< tableau >::=$	$< ident\_tableau >$ [$< taille\_tableau >$]
$< ident\_tableau >::=$	$< identificateur >$
$< liste\_instructions >::=$	$< instruction > \|\|$
	$< instruction > < liste\_instructions >$
$< instruction >::=$	$< instr\_base > \|\| < instr\_de\_controle >$
$< liste\_processus >::=$	$< processus > \ \| \ < processus >< liste\_processus >$
$< instr\_base >::=$	$< instr\_calcul > \|\| < instr\_envoi > \|\|$
	$< instr\_recoit > \|\| < instr\_creation > \|\|$
	$< instr\_test > \|\| < instr\_affectation >$
$< instr\_de\_controle >::=$	$< instr\_pour > \ \| \ < instr\_si >$
$< instr\_pour >::=$	**for** ($< instr\_affectation >$;
	$< expression >$; $< instr\_affectation >$)
	{ $< corps\_instrcontrole >$ **}**
$< corps\_instrcontrole >::=$	**{** $< instruction > \|\|$ **{** $< liste\_instructions >$ **}**
$< inst\_si >::=$	**if** ($< expression >$)
	$< corps\_instrcontrole > \ \|\|$
	if ($< expression >$)

	< *corps_instrcontrole* >
	else
	< *corps_instrcontrole* >
< *processus* > ::=	**process** < *identificateur* > (< *liste_def_param* >)
	< *corps_proc* > ||
	process < *identificateur* > ( )
	< *corps_proc* >
< *liste_def_param* > ::=	< *variable* > || < *variable* > , < *liste_def_param* >
< *instr_calcul* > ::=	**compute** (< *quantite_de_calculs* >);
< *instr_envoi* > ::=	**send** (< *destinataire* >, < *type_mess* >, < *liste_param* >); ||
	send (< *destinataire* >, < *type_mess* >, < *volume_comm* >);
< *instr_recoit* > ::=	**recv** (< *emetteur* >, < *type_mess* >); ||
	recv (< *emetteur* >, < *type_mess* >, < *liste_variables* >);
< *instr_creation* > ::=	< *variable* > =
	spawn (< *nom_processus* >, < *liste_param* >); ||
	< *variable* > = **spawn** (< *nom_processus* >);
< *instr_test* > ::=	**nrecv** (< *emetteur* >, < *type_message* >);
< *liste_param* > ::=	< *parametre* > || < *parametre* > , < *liste_param* >
< *parametre* > ::=	< *variable* > || < *valeur* > || < *id_processus* >
< *instr_affectation* > ::=	< *variable* > = < *valeur_affectation* >;
< *instr_test* > ::=	**nrecv** (< *emetteur* >,
< *liste_variables* > ::=	< *variable* > || < *variable* >, < *liste_variables* >
< *variable* > ::=	< *identificateur* > || < *tableau* >
< *valeur_affectation* > ::=	< *variable* > || < *valeur* > || < *expression* >
< *destinataire* > ::=	< *id_processus* >
< *emetteur* > ::=	< *id_processus* > || < *processus_indefini* >
< *id_processus* > ::=	< *variable* > || **parent** || **mytid**
< *processus_indefini* > ::=	**nktid**
< *type_mess* > ::=	**data** || **nktype** || < *identificateur* >
< *identificateur* > ::=	< *lettre* >< *identificateur* > || < *lettre* >
< *lettre* > ::=	**A** || **B** || ... || **Z**
< *taille_memoire* > ::=	< *valeur* >
< *taille_tableau* > ::=	< *valeur* >
< *numero_processus* > ::=	< *valeur* >
< *volume_comm* > ::=	< *valeur* >
< *quantite_de_calculs* > ::=	< *valeur* >
< *valeur* > ::=	**0** || **1** || .... || **9999999**
< *expression* > ::=	< *expression_simple* > ||
	< *expression_simple* > < *oprel* > < *expression_simple* >
< *oprel* > ::=	== || != || < || > || <= || >=
< *expression_simple* > ::=	$\{< signe >\}^{0\|1} < terme > \{< opadd >< terme >\}^*$

$< opadd > ::=$	$+ \mid\mid - \mid\mid or$
$< signe > ::=$	$+ \mid\mid -$
$< terme > ::=$	$< facteur > \{< opmult >< facteur >\}^*$
$< opmult > ::=$	$* \mid\mid / \mid\mid \% \mid\mid and$
$< facteur > ::=$	$< valeur > \mid\mid < variable > \mid\mid$
	$! < facteur > \mid\mid (< expression >)$

3.9 Exemples de programmes dans LASSIMAD

Le premier exemple est un programme parallèle qui contient deux processus principaux, un qui joue le rôle de producteur et un autre qui sera le consommateur. Le modèle producteur-consommateur permettant de représenter des comportements parallèles ainsi que les interactions d'échange de messages entre deux processus d'un programme, est un bon exemple pour être implanté en utilisant les instructions du langage LASSIMAD. Les processus qui implémentent ce programme Producteur-Consommateur écrits en langage LASSIMAD se trouvent sur la figure 3.3. La figure 3.2 montre le graphes de création et de communications de cet exemple.

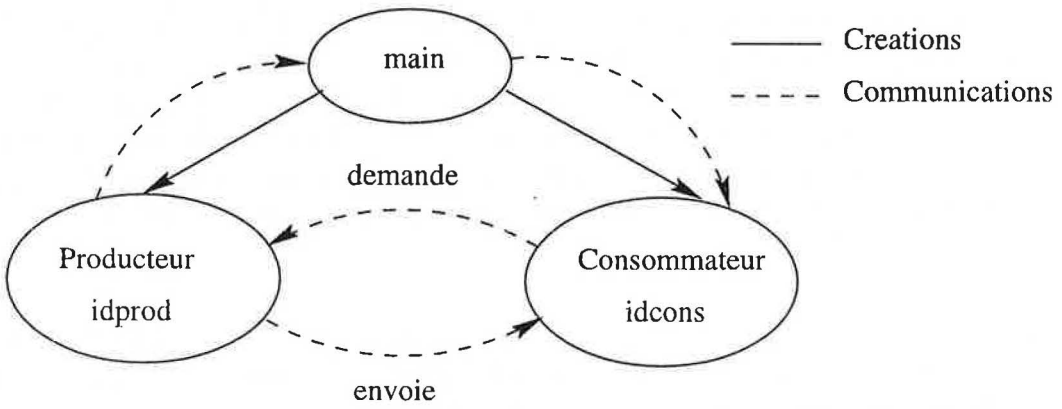


FIG. 3.2 - *Graphe du Producteur-Consommateur*

Le processus principal **main** crée d'abord un processus **Producteur** dont l'identificateur est gardé dans idprod. Pour être sûr que le processus **Producteur** a été déjà placé sur un des noeuds de la machine, le processus **main** reste en état d'attente d'un message provenant du **Producteur**. Il existe deux raisons pour lesquelles cette vérification doit être faite; la première est qu'un processus doit être placé sur un noeud

avant de lui envoyer un message et la deuxième est pour éviter l'arrêt prématuré de la simulation. Si aucun processus n'est placé sur la machine la simulation s'arrêtera (interprété comme : il n'y a plus rien à exécuter). L'arrêt de la simulation pourrait arriver si le processus **main** ne fait que les deux créations et ensuite finit son exécution; dans le cas où le placement des processus **Producteur** et **Consommateur** est en train de s'effectuer mais qu'il n'est pas complètement terminé le simulateur arrête l'exécution. Il est donc nécessaire qu'au moins un processus soit placé sur la machine.

Après avoir reçu le message provenant du processus **Producteur** le processus **main** crée le deuxième processus **Consommateur** gardant l'identificateur sur la variable **idcons** et il finit son exécution. Un paramètre commun aux deux processus **Producteur, Consommateur** est le nombre de produits à fabriquer et à consommer (4 pour ce premier exemple). Le processus **Consommateur** a un paramètre additionnel qui indique l'identificateur du Producteur auquel il va demander des produits.

D'autre part le processus **Producteur** commence son exécution en envoyant un message de type **typeSynch** au processus **main** pour lui annoncer son placement. Ensuite il reste en état d'attente d'un message dont le type implique la requête d'un produit **typeReq**. Quand le message arrive, la variable **nktid** prend la valeur de l'identificateur du processus qui a envoyé le message, et la variable **idconsreq** prend la valeur de l'identificateur du processus **Consommateur** qui fait la requête du produit. Dans cet exemple les deux variables **nktid** et **idconsreq** ont toujours la même valeur mais leur définition est nécessaire pour permettre l'interaction de plusieurs **Producteurs** et **Consommateurs** comme on le montrera dans d'autres exemples.

Pour fabriquer un produit (ou un service), le processus producteur utilise le processeur pendant un certain temps (**compute(2500)**), ensuite il l'envoie au **Consommateur (idcons)**. Si le nombre de produits à fabriquer n'est pas atteint ($nb\_fabric < nb\_produits$), le **Producteur** envoie un deuxième message au processus **Consommateur** pour lui indiquer qu'il reste de produits à fabriquer (type du message = **typeContinue**) et ensuite le **Producteur** reste à nouveau bloqué en attente d'une autre requête de produit. D'autre part, si le processus **Producteur** a fabriqué le dernier produit, il envoie un message de type **typeFin** au processus **Consommateur** et il finit ensuite son exécution.

Concernant l'exécution du processus **Consommateur**, comme dans le cas du processus **Producteur**, d'abord il envoie un message au processus **main** pour confirmer

```
list messages
    typeSynch, typeReq, typeContinue, typeFin

main
memory = 500;
var idcons, idprod;
{
   idprod = spawn(Producteur, 4);
   recv(idprod, typeSynch);
   idcons = spawn(Consommateur, idprod, 4);
   recv(idcons, typeSynch);
}
```

```
process Producteur(nb_produits)
memory = 1200;
var i, idconsreq, nb_fabric;
{
   send(parent, typeSynch);
   nb_fabric = 0;
   for(i = 0; i < nb_produits; i = i + 1)
   {
    compute(2500);  /* Production */
   recv(nktid, typeReq, idconsreq);
    send(idconsreq, data, 1500);
    nb_fabric = nb_fabric + 1;
    if(nb_fabric < nb_produits)
        send(idconsreq, typeContinue);
    else
        send(idconsreq, typeFin);
   }
}
```

```
process Consommateur(idProd, nb_produits)
memory = 2000;
var i, id, nb_consom;
{
   send(parent, typeSynch);
   nb_consom = 0;
   send(idProd, typeReq, mytid);
   for(i = 0; i < nb_comm; i = i + 1)
    {
     recv(nktid, data);
     id = nktid;
     nb_consom = nb_consom + 1;
     recv(id, nktype)
     if(nb_consom < nb_produits)
     {
        if(nktype == typeContinue)
          send(id, typeReq, mytid);
     }
     compute(3000); /* Consommation */
    }
}
```

FIG. 3.3 - *Processus Producteur-Consommateur*

son placement. Le **Consommateur** prend en compte un **Producteur** initial **idprod** (paramètre au moment de sa création) pour lui faire des requêtes de produits par l'envoi d'un message de type **typeReq**. Dans le même message le **Consommateur** envoie un paramètre contenant son propre identificateur (**mytid**). A partir de ce moment le Consommateur peut recevoir le produit provenant d'un Producteur quelconque gardant son identificateur sur la variable **nktid**.

Une fois que le **Consommateur** a reçu le produit il reste en état d'attente d'un autre message du même **Producteur** pour savoir s'il lui reste ou non d'autres produits à fabriquer. Si le nombre de produits consommés est inférieur au nombre de produits devant être consommés et si le Producteur qui vient de lui envoyer un produit a d'autres produits à fabriquer, le processus **Consommateur** lui envoie une nouvelle requête. D'autre part, si le processus **Consommateur** a utilisé tous les produits qu'il devait consommer, il finit son exécution. Pendant que le prochain produit est fabriqué le consommateur consomme le produit reçu, l'utilisation ou consommation du produit demande aussi un temps d'utilisation du processeur (instruction **compute(3000)**).

Ce premier exemple se limite seulement à lier un consommateur avec un producteur. Cependant si l'on prend en compte que les vitesses de production et de consommation ne sont pas toujours synchronisées, on pourrait considérer des systèmes où il existe plusieurs consommateurs et plusieurs producteurs. Les exemples suivants montrent les programmes dans le langage LASSIMAD qui représentent ces types de systèmes.

En utilisant les mêmes processus de base **Producteur** et **Consommateur** définis dans le premier exemple, considérons un autre programme qui crée plusieurs processus consommateurs qui demandent les produits d'un seul producteur. Cela serait le résultat d'une fabrication trop rapide par rapport au temps de consommation du produit (la quantité de calculs pour consommer le produit serait donc plus grande que la quantité de calculs pour le fabriquer). Le processus principal de ce deuxième programme est montré sur la figure 3.4 et ses graphes de création et de communications sur la figure 3.5.

Dans ce deuxième exemple le processus **main** utilise un vecteur pour garder les identificateurs de 3 processus **Consommateurs** (**idcons**[3]), la variable **idprod** contient

```
list messages
    typeSynch, typeReq, typeContinue, typeFin

main
memory = 600;
var idcons[3], idprod, i;
{
   idprod = spawn(Producteur, 15);
   recv(idprod, typeSynch);
   for(i = 0; i < 3; i = i + 1)
      idcons[i] = spawn(Consommateur, idprod, 5);
   for(i = 0; i < 3; i = i + 1)
      recv(nktid, typeSynch);
}
```

FIG. 3.4 - *Processus principal du programme 1Producteur-3Consommateurs*

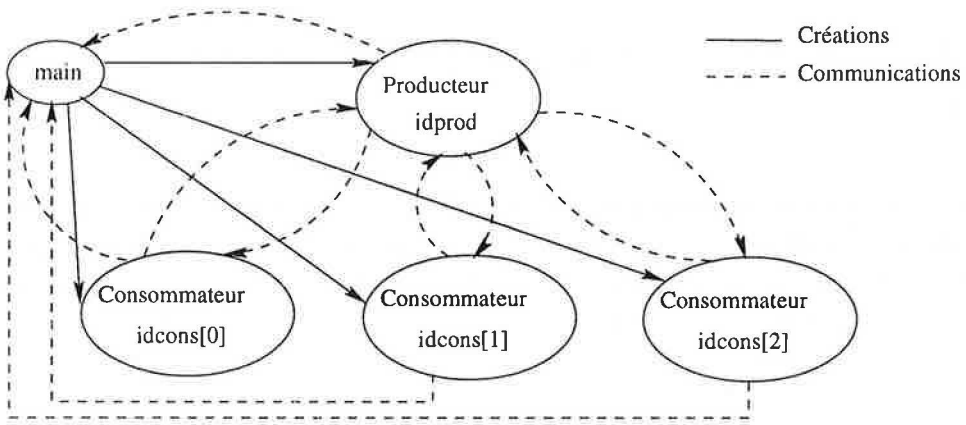


FIG. 3.5 - *Graphe: 1 Producteur, 3 Consommateurs*

l'identificateur du seul processus **Producteur**. L'exécution commence comme dans l'exemple précédent; d'abord le processus **main** crée le processus **Producteur** (qui fabriquera 15 produits) et il attend un message indiquant son placement. Ensuite, à l'aide d'une boucle, il crée les trois processus **Consommateurs** ayant tous les mêmes paramètres de création (identificateur du seul producteur **idprod** et 5 produits à demander). De cette manière le producteur fabrique 15 produits et chacun des consommateurs en demande 5.

```
list messages
    typeSynch, typeReq, typeContinue, typeFin

main
memory = 70;
var idcons, idprod[3], i;
{
   for(i = 0; i < 3; i = i + 1)
      idprod[i] = spawn(Producteur, 5);
   for(i = 0; i < 3; i = i + 1)
      recv(nktid, typeSynch);
   idcons = spawn(Consommateur, idprod[0], 15);
   recv(idcons, typeSynch);
   for(i = 1; i < 3; i = i + 1)
      send(idprod[i], typeReq, idcons);
}
```

FIG. 3.6 - *Processus principal du programme 3Producteurs-1Consommateur*

Un troisième exemple concerne le cas contraire : la vitesse de consommation est plus grande que la vitesse de production. On trouve sur la figure 3.6 le processus principal de ce troisième programme et sur la figure 3.7 les graphes de créations et de communications associés.

Le processus **main** utilise cette fois-ci un vecteur pour garder les identificateurs des 3 processus producteurs (**idprod[3]** et la variable **idcons** pour garder l'identificateur du seul consommateur. D'abord les trois processus **Producteur** sont créés en ayant

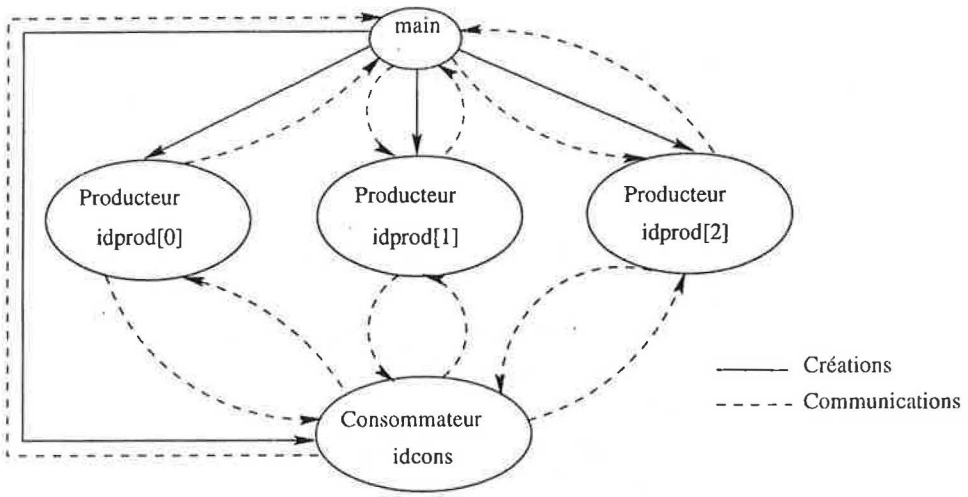


FIG. 3.7 - *Graphe : 3 Producteurs, 1 Consommateur*

comme paramètre le même nombre de produits à fabriquer (5 pour cet exemple). Après avoir vérifié le placement des trois processus créés, le processus **main** crée le seul processus consommateur ayant comme paramètre le premier producteur créé (**idprod[0]**) et le nombre de produits à consommer égal à 15. Puisque le processus **Consommateur** va d'abord envoyer ses requêtes au seul processus **Producteur idprod[0]**, le processus **main** doit activer les deux autres producteurs pour qu'ils envoient leurs produits au consommateur **idcons**. A cet effet le processus main fait deux requêtes de produits aux producteurs **idprod[1]** et **idprod[2]** mais en envoyant l'identificateur du consommateur (**idcons**) dans le paramètre qui indique l'identificateur du consommateur qui fait la requête (**idconsreq**).

Finalement, les figures 3.8 et 3.9 montrent le processus **main** et les graphes de créations et de communications du dernier programme qui combine les deux programmes précédents. De la même manière on pourrait construire des programmes plus complexes dont le graphe principal de créations est un arbre et chacun des noeuds (processus) est défini comme s'il était ce dernier processus **main** qui modélise l'interaction de N producteurs avec M consommateurs.

Etant donné que le but principal de SIMAD est de permettre la comparaison des algorithmes de placement dynamique, on s'est intéressé surtout à construire des

```
list messages
    typeSynch, typeReq, typeContinue, typeFin

main
memory = 70;
var idcons[3], prod, idprod[3], cons, i;
{
   prod = spawn(Producteur, 15);
   recv(prod, typeSynch);
   for(i = 0; i < 3; i = i + 1)
      idcons[i] = spawn(Consommateur, prod, 5);
   for(i = 0; i < 3; i = i + 1)
      recv(nktid, typeSynch);
   for(i = 0; i < 3; i = i + 1)
      idprod[i] = spawn(Producteur, 5);
   for(i = 0; i < 3; i = i + 1)
      recv(nktid, typeSynch);
   cons = spawn(Consommateur, idprod[0], 15);
   recv(cons, typeSynch);
   for(i = 1; i < 3; i = i + 1)
      send(idprod[i], typeReq, cons);
}
```

FIG. 3.8 - *Processus principal du programme NProducteurs-MConsommateurs*

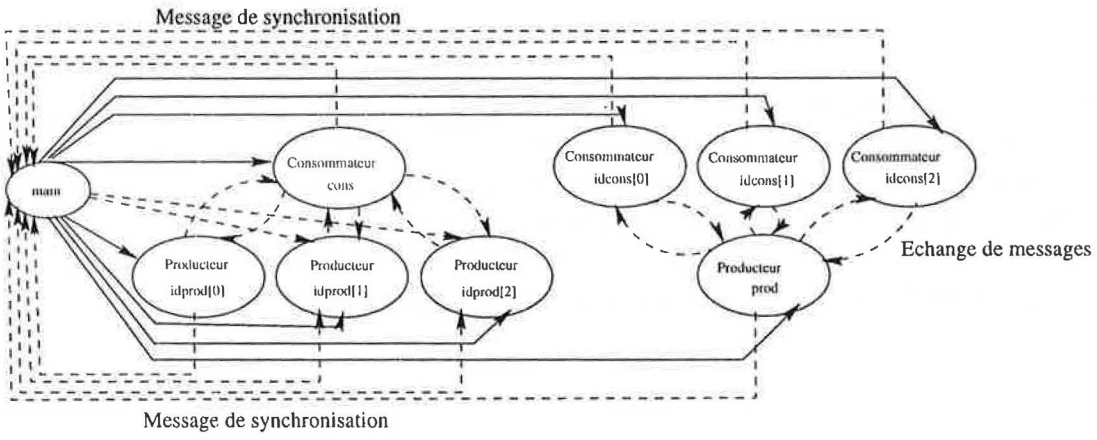


FIG. 3.9 - *Graphe N Producteurs M Consommateurs*

programmes ayant des exécutions déterministes pour que les comparaisons soient valables. Mais si l'utilisateur veux étudier le comportement du système parallèle exécutant des applications dont le nombre de créations est déterminé au cours de l'exécution il doit vérifier que l'exécution se termine à un moment donnée.

3.10 L'ensemble de programmes test

Pour terminer ce chapitre nous présentons les programmes écrits en LASSIMAD utilisés pour construire des cas test qui aident à l'évaluation des algorithmes de placement dynamique. Ces programmes sont ceux qui s'utiliseront dans le dernier chapitre qui présente des tests et des comparaisons. Nous avons regroupés les programmes en 2 classes : les programmes où les processus communiquent peu et les programmes où les processus échangent plusieurs messages.

3.10.1 Les programmes où les processus communiquent peu

Ce premier groupe contient des programmes dans lesquels la parallélisation des calculs est importante. Pour chaque programme, les processus font une certaine quantité de calculs avant d'envoyer un message de terminaison, soit à leurs pères, soit à un processus qui attend la terminaison des processus créés.

Dans les programmes la quantité de calculs à effectuer par chaque processus est déterminée de plusieurs façons. Pour expliquer ces comportements nous avons séparé les programmes en 2 sous-groupes : les programmes qui ont un graphe de créations en forme d'arbre complet et les programmes avec d'autres graphes de créations.

a) Les programmes avec un graphe de création en forme d'arbre complet

Les programmes de ce premier sous-groupe ont une hauteur H associée. Le processus racine a le niveau 1, les processus créés par la racine ont le niveau 2 et ainsi de suite jusqu'aux processus feuilles qui ont le niveau H. Un processus qui n'est pas une feuille crée N (où N est une constante) autres processus.

La table suivant montre la liste des programmes dans ce sous-groupe où les processus communiquent peu et qui ont un graphe de création en forme d'arbre complet. Chacun des programmes a ses propres caractéristiques par rapport à la quantité de

calculs effectués.

Num	Nom du programme	Annexe
1	processus-indep.prg	A.1
2	arbre-complet-1.prg	A.2
3	prg-all-1.prg	A.3
4	pparallel.prg	A.4
5	travaille-tjrs.prg	A.5
6	arbr1-AuDiAuDi.prg	A.6

- 1.- Le programme processus-indep.prg
  Les processus de ce programme font tous la même quantité de calculs. Dès qu'un processus fini ses calculs il envoie un message au processus "initial" et ensuite finit son exécution. Le processus "initial" est celui qui se charge de créer au processus racine de l'arbre et de recevoir les messages de terminaison de tous les processus de l'arbre. Nous pouvons voir les graphes de créations et de communications sur la figure 3.10.

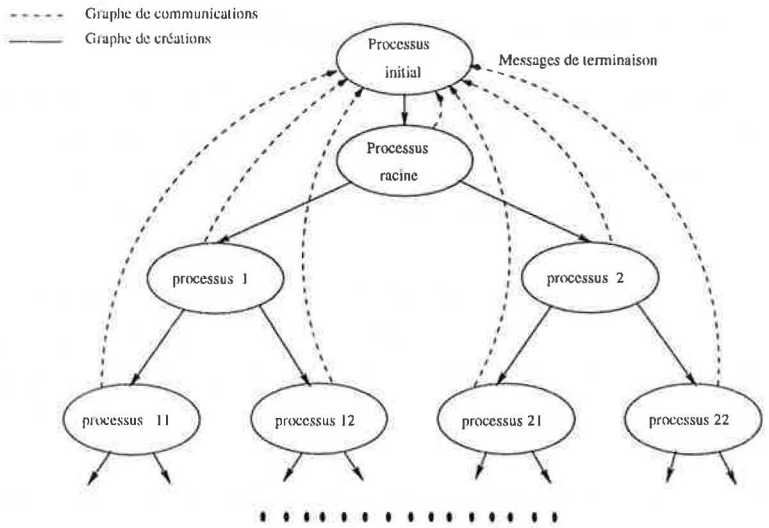


FIG. 3.10 - *Graphes de communicaitons et de créations du programme processus-indep.prg*

Ce programme est intéressant du fait que la valeur de la quantité de calculs à réaliser influence le placement des processus; si la quantité de calculs à réaliser n'est pas importante, un processus finira plus vite son exécution et un noeud

pourra exécuter plusieurs processus. D'autre part, si les processus font une quantité importante de calculs, un bon algorithme de placement cherchera à placer le moins de processus sur un seul noeud.

- 2.- Le programme arbre-complet-1.prg
  Comme dans le cas précédent, les processus de ce deuxième programme font tous la même quantité de calculs. La différence est la destination du message de terminaison qui sera dirigé vers les processus pères au lieu du processus initial. Un processus père crée d'abord ses fils, puis fait ses calculs, ensuite se bloque en attendant l'arrivée des messages de terminaison de ses fils et finalement envoie lui même un message de terminaison à son père (s'il n'est pas la racine). Un processus qui n'est pas père fait ses calculs et envoie ensuite le message de terminaison à son père. Les graphes de communications et de créations de ce programme sont illustrés sur la figure 3.11.

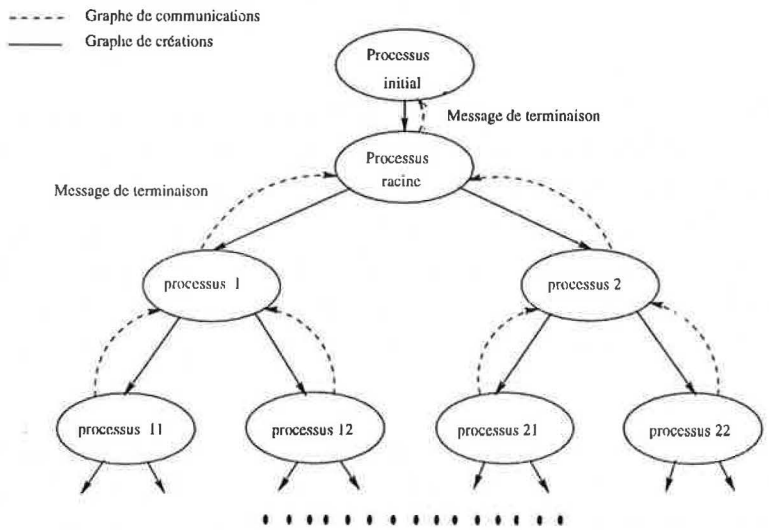


FIG. 3.11 - *Graphes de communicaitons et de créations du programme arbre-complet-1.prg*

L'intérêt de ce deuxième type de programme est qu'il nous permet d'avoir, en même temps, plus de processus sur la machine, indépendamment de la quantité de calculs à réaliser par les processus. L'idée de synchroniser la terminaison d'un processus avec la terminaison de ses fils garantit qu'au moins H processus, appartenant à une branche de l'arbre (de hauteur H), seront présents sur la machine à un moment donné.

- 3.- Le programme prg-all-1.prg
  Le troisième programme a un graphe de créations et de communications comme ceux du deuxième programme (figure 3.11). Dans le cas précédent, puisque tous les processus font la même quantité de calculs, les processus qui sont plus près de la racine vont normalement finir plus vite leur exécution que ceux qui se trouvent vers les feuilles. Les processus ayant un niveau plus petit dans l'arbre vont passer plus de temps à attendre l'arrivée des messages des processus avec de niveaux plus grands.
  Pour diminuer cette attente des messages, nous avons construit le troisième programme "prg-all-1.prg" où les quantités de calcul effectuées par les processus sont décroissantes selon le niveau qu'ils ont dans l'arbre. Les processus appartenant au niveau X + 1, (X = 1..H-1) font une quantité de calculs inférieure à celle effectuée par les processus dans le niveau X. La quantité de calculs QC effectuée par un noeud (N) au niveau X+1, est la même que celle effectuée par un noeud au niveau X moins une quantité Q2 de calculs :

$$QC(N_{X+1}) = QC(N_X) - Q2$$

- 4.- Le programme pparallel.prg
  Ce quatrième programme est une variation du programme 3 (avec les mêmes graphes de créations et de communications que sur la figure 3.11). Pour ce programme chacun des processus père travaille pendant un temps équivalent au temps de travail de ses fils plus le temps des communications qu'ils font. Les temps sont calculés selon les caractéristiques d'une machine donnée (capacité de calcul et débit des liens) et en supposant qu'on place un processus par noeud et que les noeuds de la machine sont complètement connectés.

- 5.- Le programme travaille-tjrs.prg
  Nous avons fait le cinquième programme (avec le même graphe de communications et de créations que ceux montré sur la figure 3.11) qui nous permet d'étudier l'influence des distances dans la machine entre les sites des processus pères et ceux de leurs fils. Un processus père, après avoir créé ses fils répète une certaine quantité Q1 de calculs jusqu'à recevoir les messages de terminaison de tous ses fils. Un processus qui n'est pas père, fait Q2 calculs et envoie ensuite un message de terminaison à son père.
  Dans les programmes que nous avons construit, nous donnons une valeur plus petite à Q1 par rapport à celle donnée à Q2. Le but de ces tests est de déterminer par comparaison quel est l'algorithme de placement qui a fait exécuter le programme avec le temps d'exécution le plus petit et en utilisant le plus de noeuds.

- 6.- Le programme arbr1-AuDiAuDi.prg
  Le dernier programme dans ce sous-groupe contient un processus initial qui se

charge de générer de la charge sur une machine en exécutant un programme X. Lorsque les noeuds finissent l'exécution de X le processus initial lance à nouveau l'exécution du programme X. Le programme X que nous utilisons a la forme du troisième programme prg-all-1.prg présenté antérieurement.
L'objectif de ce programme de test est d'étudier le comportement des algorithmes de placement vis à vis de l'actualisation de l'information sur l'état de charge possédée par chacun des noeuds. L'information est actualisé: lorsque la charge augmente, lorsqu'elle diminue petit à petit jusqu'à laisser un seul processus sur la machine et lorsque à nouveau elle augmente pour la deuxième exécution du programme.

b) Les programmes avec d'autres graphes de créations

Dans le second sous-groupe de programmes, où les processus font peu de communications, les graphes de créations sont: soit un arbre de hauteur 2, soit un arbre irrégulier de hauteur H. Ces programmes permettent de comparer les performances des algorithmes de placement sous différents graphes de créations des processus.
La table suivante montre la liste des programmes qui appartiennent à ce sous-groupe:

Num	Nom du programme	Annexe
7	p1pere-Nfils.prg	A.7
8	p1pere-Nfils-sync.prg	A.8
9	disques.prg	A.9
10	reines.prg	A.10
11	arbre-tableau.prg	A.11

- 7.- Le programme p1pere-Nfils.prg
  Dans ce cas (programme 7) un seul processus crée N autres processus. Les N processus font la même quantité de calculs et à la fin de l'exécution ils envoient un dernier message au processus père. Les graphes de créations et de communications pour ce programme sont présentés dans la figure 3.12.

Ce type de programme permet d'étudier le comportement d'un algorithme de placement lorsqu'un seul processus génère tous les autres processus du programme. Dans le cas où la connaissance d'un noeud sur l'état de charge de la machine est partielle, la vitesse d'actualisation de cette connaissance est importante.

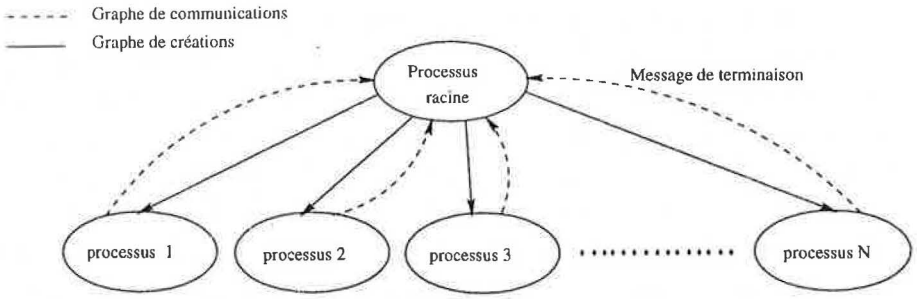


FIG. 3.12 - *Graphes de communicaitons et de créations du programme p1pere-Nfils.prg*

- 8.- Le programme p1pere-Nfils-sync.prg
  Le huitième programme a le même graphe de créations que le programme précédent et les processus font tous la même quantité de calculs. La différence avec le programme 8 est que les processus créés sont synchronisés avant de réaliser leurs calculs; lorsqu'un processus commence son exécution il envoie un message au processus racine (pour lui annoncer son placement réel), ensuite il attend une réponse qui est un message indiquant que les calculs peuvent être effectués. A la fin de l'exécution les processus envoient un dernier message de terminaison au processus racine.
  D'autre part le processus racine, après avoir créé N processus, se bloque dans l'attente de N messages provenant de ses fils. Une fois que le processus racine est sûr que ses fils sont physiquement placés, il leur envoie un message pour qu'ils commencent leurs calculs. Finalement le processus racine attend les derniers messages de terminaison de ses fils. Les graphes de créations et de communications de ce programme sont presque les mêmes que ceux du programme précédent (figure 3.12), la seule différence est que les arcs du graphe de communications sont bidirectionels (pour la transmission des messages de synchronisation).

  Dans le programme précédent, un processus pouvait finir son exécution sans que le processus racine ait fini de créer tous les processus ou bien sans que les noeuds créés soient placés sur un noeud. Grâce à la synchronisation des processus ce programme de test nous permet d'avoir, à un moment donné, tous les processus placés sur les noeuds de la machine. Les algorithmes de placement pourront être donc comparés selon la distribution de charge obtenue sur la machine.

- 9.- Le programme disques.prg
  Pour comparer les algorithmes de placement exécutant de programmes avec

d'autres types de graphes de créations, nous avons construit le programme "disques.prg" où chaque processus peut créer au maximum N autres processus. Le graphe de créations de ce programme est un arbre irrégulier qui représente un arbre d'espaces d'états du problème des N disques :
Il y a N disques, chacun d'eux est divisé en M secteurs, dans chaque secteur il y a un numéro entier. Nous devons faire tourner les N disques de telle sorte que la somme des valeurs du même secteur de tous les disques soit inférieure ou égale à un numéro K. Pour ce programme nous avons fixé le numéro de secteurs M = 4.
Dans ce programme les calculs effectués par les processus ne sont pas importants. Le processus racine, à la fin de l'exécution, fait une quantité de calculs égale au numéro de solutions trouvées.

- 10.- Le programme reines.prg
  Le dixième programme possède aussi un graphe de créations irrégulier qui représente l'arbre d'espaces d'états du problème de placer N reines sur un tableau avec NxN cases. Les reines doivent être placées sur certaines cases valables, de telle sorte qu'elles ne puissent pas s'éliminer entre elles.
  Dans ce cas, seulement les processus qui trouvent une case valable pour placer une reine font des calculs. La quantité de calculs à effectuer dépend des positions (x,y) de la case du tableau. Si un processus trouve une case valable dans la position (x,y) et si la première reine placée sur la première ligne se trouve dans la position (x1,1) le noeud fait $(x1 * 100000) + (x * 1000) + y$ calculs.

- 11.- Le programme arbre-tableau.prg
  Le dernier programme de ce sous-groupe a le même comportement que le programme 4 "pparallel.prg" du premier sous-groupe. Chacun des processus père travaille un temps équivalent au maximum des temps de travail et de communication faits par ses fils. La différence est que dans le programme "pparallel.prg" le nombre de créations effectuées par un processus père était un numéro N, tandis que dans le programme "arbre-tableau.prg" le nombre de créations effectuées par un processus dépend du niveau qu'il a dans l'arbre.
  Pour implémenter cet algorithme nous avons utilisé un tableau t de taille H où H est la hauteur de l'arbre. Chaque élément t[i] (i=1..H), contient le numéro de créations à effectuer par les noeuds qui ont le niveau i dans l'arbre. Comme exemple, nous pouvons voir sur la figure 3.13 les graphes de créations et de communications d'un programme de hauteur 4 avec un tableau t= 3,1,2,0.

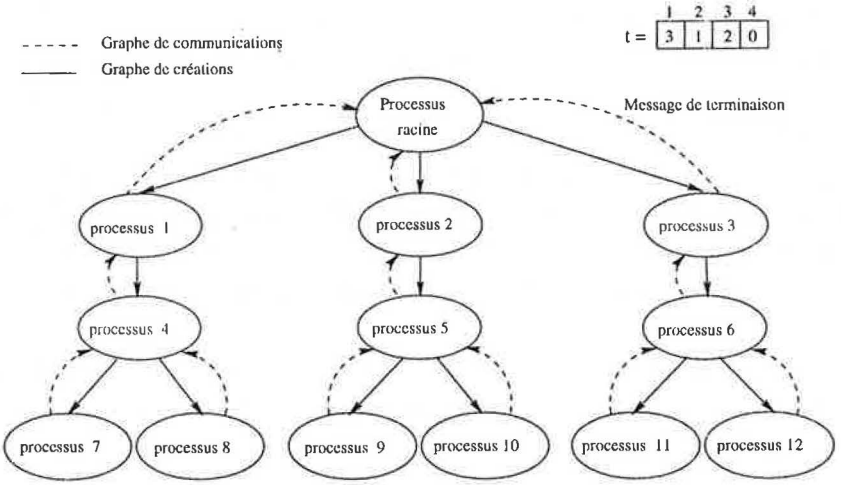


FIG. 3.13 - *Graphes de communicaitons et de créations du programme arbre-tableau.prg*

3.10.2 Les programmes où les processus font plusieurs échanges de messages

Si les processus d'un programme parallèle échangent des données fréquemment et s'ils sont placés sur des sites très éloignés dans la machine, le temps d'exécution du programme peut être détérioré. Cela est une conséquence de la surcharge des liens de communications utilisés pour faire arriver le message du noeud source au noeud destinataire.

Avec les programmes qui appartiennent à ce deuxième groupe nous générons des charges sur les liens de communication de la machine de trois manières : en augmentant le volume des données à transmettre, en augmentant le nombre de messages échangés entre deux processus, et en ayant des graphes de communications généraux qui ne relient pas seulement les pères avec leurs fils.

Comme dans le cas du premier groupe, nous avons divisé ce second grupe en deux autres sous-groupes : les programmes avec un graphe de créations en forme d'arbre complet et les programmes avec d'autres graphes de créations. Dans les pargraphes suivants nous allons présenter ces deux sous-groupes.

a) Les programmes avec un graphe de création en forme d'arbre complet

Comme nous avions dit précédemment, un programme dans ce sous-groupe possède des processus qui font tous le même nombre de créations (sauf les feuilles de l'arbre).

La définition d'un programme est donnée par le numéro de créations à effectuer par chaque processus et par la hauteur H de l'arbre.
La différence entre ces programmes et ceux du premier groupe est le nombre de communications effectuées et les processus avec lesquels ils communiquent.

La table suivante montre ce premier sous-groupe de programmes où les processus font plusieurs échanges de messages et qui ont un graphe de créations en forme d'arbre complet.

Num	Nom du programme	Annexe
12	arbre-complet-2.prg	B.1
13	prg-comm-1.prg	B.2
14	comm3FFH7.prg	B.3

D'abord nous présentons deux programmes où les processus pères communiquent avec ses fils et ensuite un programme où les communications se font entre processus frères.

- 12.- Le programme arbre-complet-2.prg
  Dans le premier programme de ce sous-groupe possédant un graphe de créations en forme d'arbre complet, les processus pères font N échanges de messages avec chacun de leurs processus fils. Un échange est composé de deux envois de messages : un du processus père vers son fils et un autre du fils vers son père. Les processus font une même quantité de calculs entre un envoi et une réception de message. Lorsque les processus fils terminent leur exécution, ils envoient un dernier message vers leurs pères. La figure 3.14 illustre les graphes de créations et de communications de ce programme.

  Le programme "arbre-complet-2.prg" est une variation du programme 2 ("arbre-complet-1.prg"), mais dans ce cas les communications fréquentes faites entre père et fils influencent plus le temps total d'exécution. Un bon algorithme de placement essayera d'utiliser au mieux les liens de communications de la machine sans provoquer d'encombrements.

- 13.- Le programme prg-comm-1.prg
  Ce programme est une variation du programme 3 ("prg-all-1.prg") avec les mêmes graphes de créations et de communications (figure 3.14). Les processus qui sont dans le niveau X de l'arbre font plus de calculs que ceux qui sont dans le niveau X+1 (X=1..H-1). Comme dans le programme précédent ("arbre-complet-2.prg") nous avons ajouté plusieurs échanges de messages entre les

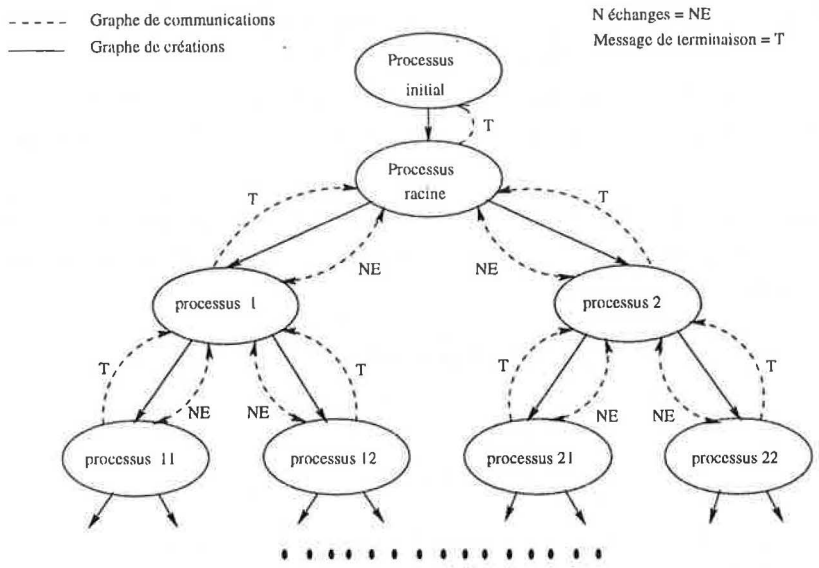


FIG. 3.14 - *Graphes de créations et de communications du programme arbre-complet-2.prg*

processus pères et leurs fils. Chacun des processus pères fait N échanges de messages avec ses fils, entre chaque envoi et réception les processus effectuent la quantité de calculs relative à leur niveau dans l'arbre.

- 14.- Le programme comm3FFH7.prg
  Le programme "comm3FFH7.prg" possède un graphe de créations en forme d'arbre complet et un graphe de communications plus général. Dans ce programme les processus créés par le même père (frères) font N échanges de messages et entre chaque envoi et réception font une quantité de calculs. Les identificateurs des processus frères sont envoyés par le processus père vers ses fils une fois que le fils ont été placés.

  Après avoir fini leur exécution, les processus fils envoient un dernier message de terminaison à leurs pères. Nous pouvons voir sur la figure 3.15 les graphes de créations et de communications associés.

Ce programme nous permet de comparer les algorithmes de placement exécutant des programmes avec un graphe de communications plus complexe. Si les sites de placement de deux processus frères sont trop éloignés, les messages

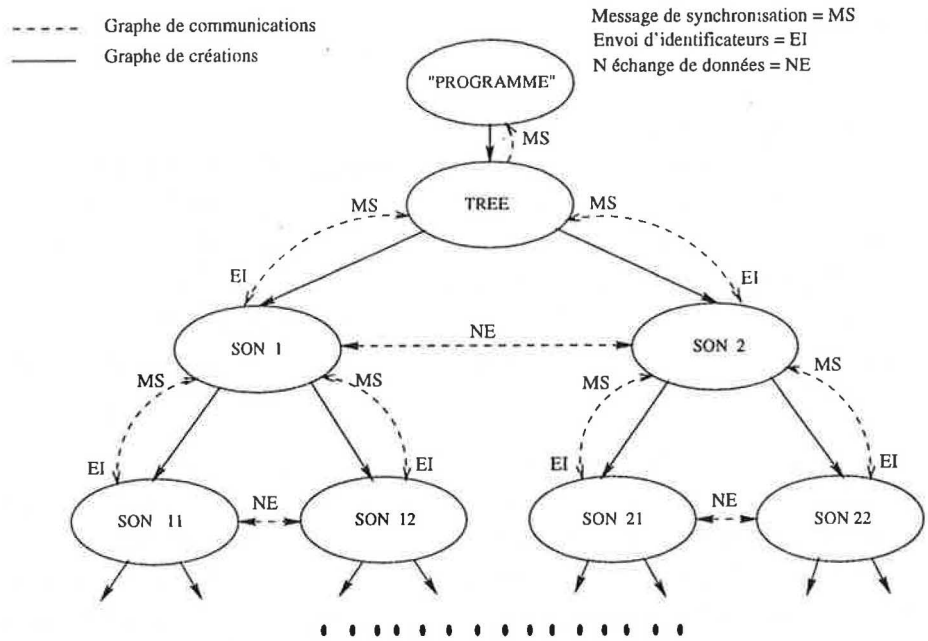


FIG. 3.15 - *Graphes de créations et de communications du programme comm-3FFH7.prg*

vont prendre plus de temps en arriver à leur destination et le temps total d'exécution augmentera.

b) Les programmes avec d'autres graphes de créations

Le deuxième sous-groupe de programmes où les processus font plusieurs échanges de messages est composé par quatre programmes qui ont des graphes de créations et de communications un peu plus complexes. La table suivante montre la liste de programmes dans ce second sous-groupe.

Num	Nom du programme	Annexe
15	espace-d-etats.prg	B.4
16	fourier.prg	B.5
17	fourier-par.prg	B.6
18	prodN-consM.prg	B.7

Dans le premier programme nous utilisons un graphe de communications entre père et fils. Pour les trois derniers programmes, les identificateurs des processus destinataires ou récepteurs des messages sont passés en forme de paramètres ou bien par

l'envoi de messages.

- 15.- espace-d-etats.prg
  Ce programme est une variation du programme 10 ("reines.prg"). Le graphe de créations est irrégulier et représente l'arbre d'espaces d'états pour résoudre le problème de placer N reines sur un tableau avec NxN caisses. Les processus dans le niveau X de l'arbre font plus de calculs que ceux qui se trouvent dans le niveau X+1 (X=1..H-1). Les processus pères font K échanges de messages avec chacun de leurs fils, et entre chaque envoi et réception effectuent la quantité de calculs respective.

- 16 et 17.- Les programmes fourier.prg et fourier-par.prg
  Dans ces programme, un seul processus initial crée 2^n, $n \geq 1$ processus. Le processus initial attend ensuite que ses fils soient tous placés sur la machine en recevant 2^n messages de leur part. Les communications entre les processus représentent les échanges effectués pour trouver en parallèle la Transformée Rapide de Fourier [Isl94]. Sur la figure 3.16 nous avons un exemple de graphe de communication pour un programme avec $2^3 = 8$ processus.

  Les identificateurs des processus avec lesquels un processus fils communique sont calculés de deux manières :

  - Dans le programme fourier.prg Le processus principal envoie un vecteur de taille n à chacun de ses fils. Le vecteur contient les n identificateurs des processus avec lesquels un processus fils fait ses échanges de messages. Chaque processus fils attend que le processus initial lui envoie son vecteur pour commencer le travail.
  - Dans le programme fourier-par.prg Les identificateurs des processus avec lesquels un processus fils P communique sont demandés par le processus P au processus initial au cours de l'exécution.

  Ces deux programmes de test nous permettent de faire la comparaison des algorithmes de placement par l'exécution d'applications où le graphe de communications est plus complexe.

- 18.- prodN-consM.prg
  Le dernier programme dans ce sous-groupe possède un graphe de créations et de communications comme ceux montrés sur la figure 3.17. Un processus initial crée N autres processus, les N processus créés vont exécuter un programme

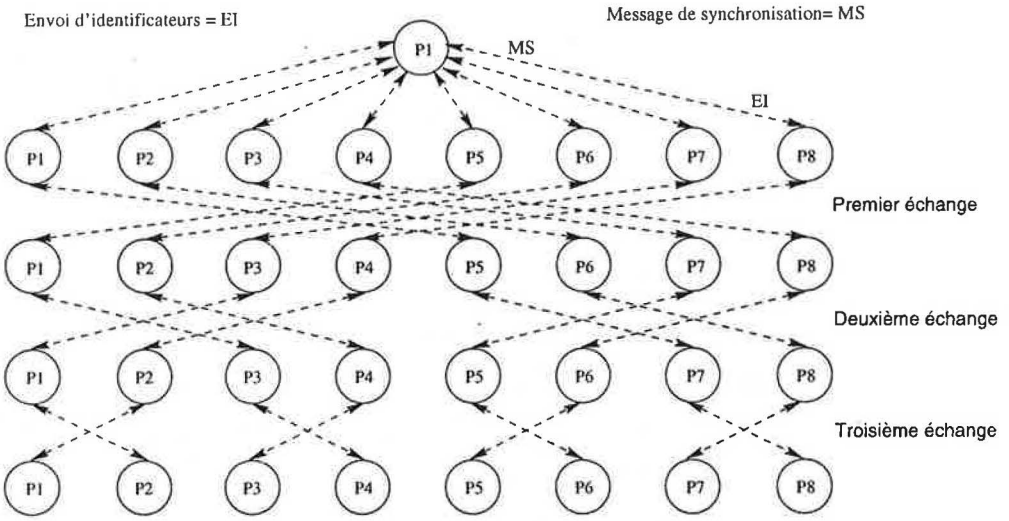


FIG. 3.16 - *Graphe de communications du programme Fourier*

qui représente un système producteur/consommateur. Les processus qui font partie de ce système sont séparés en deux groupes : d'un côté nous avons 1 producteur avec K consommateurs et d'autre côté nous avons K producteurs avec 1 consommateur.

Ce type de programme permet de décrire le comportement des système où les nombres de services et de clients sont variés. Un algorithme de placement cherchera à placer un processus producteur pas trop loin de ses processus clients.

3.11 Conclusions du chapitre

Dans ce chapitre nous avons présenté quelques types de programmes utilisés pour mesurer les performances des systèmes parallèles. Il existe des programmes réels destinés à résoudre un problème concret et des programmes synthétisés qui ont la seule tâche d'utiliser les ressources du système (comme les programmes réels), mais sans un problème particulier à résoudre.

Lorsque le but est d'étudier le comportement et d'améliorer les performances d'un système parallèle, il est intéressant d'utiliser des programmes relatifs aux problèmes

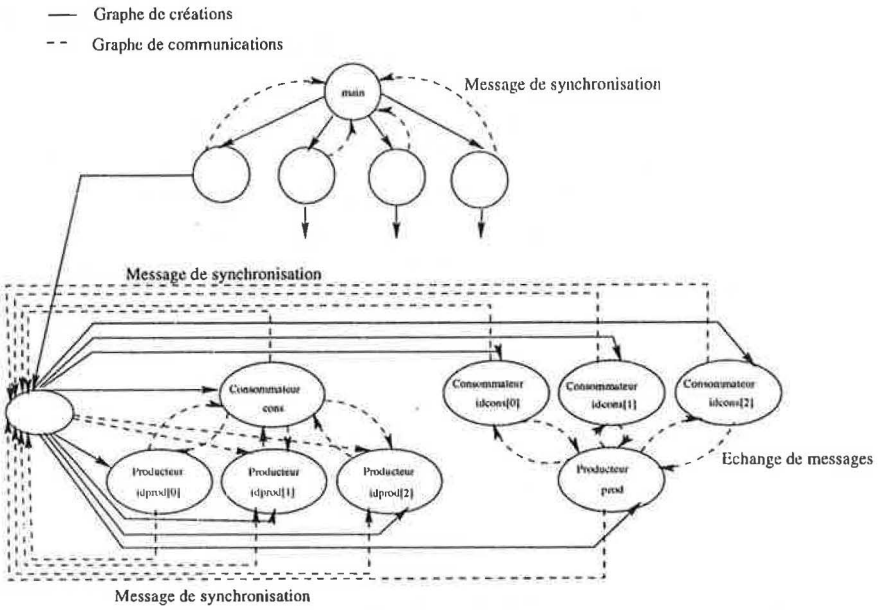


FIG. 3.17 - *Graphe de communications du programme prodN-consM*

réels. Mais il existe une autre alternative, les langages de programmation dits synthétiques qui permettent de décrire le comportement d'un programme parallèle réel en le synthétisant. Le programme est alors modélisé par une série d'instructions qui provoquent approximativement les mêmes consommations de ressources que le programme réel.

Nous avons proposé le langage LASSIMAD qui génère des programmes synthétiques utilisés par le simulateur SIMAD. Ce nouveau langage nous a permis de construire des programmes **test** qui seront utilisés pour l'étude et la comparaison des algorithmes de placement dynamique.

L'ensemble de programmes **test** est regroupé en deux parties principales : les programmes dont l'objectif est la parallélisation des calculs (à grosse granularité) et les programmes où les communications sont importantes (à granularité fine). Dans chaque groupe nous avons séparé les programmes selon le type de graphe de créations utilisé : les programmes avec un graphe de créations en forme d'arbre complet et les programmes avec des graphes de créations en forme d'arbre de hauteur 2 et d'arbre irrégulier.

Les programmes de test seront exécutés par SIMAD et un algorithme de placement dynamique décidera quels sont les sites d'exécution des processus. Nous présenterons

au chapitre 5 quelques expériences que nous avons menées, dans le but d'évaluer les performances et le comportement de notre approche du placement dynamique de processus.

Chapitre 4

Un Algorithme Evolutif pour l'équilibrage dynamique de charge

Sommaire

Comme il a été vu dans le chapitre 1, de nombreux travaux ont été réalisés sur le placement dynamique et l'équilibrage de la charge. La structure générale des algorithmes de placement dynamique fait intervenir deux éléments, le premier, identifié comme la partie qui permettra de récupérer l'information concernant l'état de charge global ou local du système (élément d'information) et le deuxième qui peut éventuellement utiliser cette information pour décider du placement d'un processus sur un autre noeud (élément de contrôle).

L'algorithme présenté dans ce chapitre est un Algorithme **Evolutif** qui se place dans le cadre de l'équilibrage dynamique de la charge d'une machine parallèle de type MIMD à mémoire distribuée, par répartition des processus. L'algorithme possède un élément d'information distribué qui permet d'appliquer un traitement évolutif sur l'information locale de chaque noeud. Il utilise certains opérateurs spéciaux qui rendent possible cette évolution. L'élément de contrôle est également distribué sur chaque noeud de la machine et il se base sur la connaissance fournie par l'élément d'information.

L'organisation de ce chapitre est la suivante, d'abord nous présentons l'environnement d'exécution de l'algorithme évolutif, puis nous expliquerons en détail l'élément d'information qui comprend la représentation de la connaissance partielle du système (Solution de Placement) et les opérateurs (croissance, déclin, remplacement et fusion) permettant l'évolution de cette information en fonction des changements de charge du système. Finalement nous parlerons de l'élément de contrôle en expliquant les algorithmes de base.

4.1 L'environnement d'exécution de l'Algorithme Evolutif

L'Algorithme Evolutif sert à placer les processus qui se créent pendant l'exécution d'une seule application ou de plusieurs applications sur une machine parallèle de type MIMD sans mémoire commune. Ce placement se fait sans avoir besoin de connaître à l'avance une estimation des relations de communication entre les processus ni des quantités de calculs à réaliser. Deux processus de différentes applications peuvent

donc être placés sur un même nœud.

L'accés à une donnée entre deux processus se fait par l'envoi et la réception de messages (modèle de programmes MIMD).

Les sites de placement des processus sont detérminés au moment de leur création et s'adaptent aux changements de l'état de charge du système. Actuellement l'algorithme ne prend pas en compte la migration de processus, une fois qu'un processus commence son exexécution sur un processeur, il ne sera plus déplacé.

L'algorithme a été étudié dans le cadre de l'exploitation de machines multiprocesseurs dédiées, c'est-à-dire qu'elles démarrent l'exécution d'une ou plusieurs applications spécifiques à un instant donné, mais avant et après leur exécution tous les nœuds ont une charge nulle.

Nous expliquerons dans la section suivante comment nous introduisons le comportement évolutif dans la recherche dynamique des sites de placement.

4.2 La connaissance partielle de l'état de la machine

L'élément d'information dans l'Algorithme Evolutif permet d'avoir une connaissance partielle de l'état de charge de la machine sur chaque nœud (information distribuée).

La charge locale d'un noeud est quantifiée par une valeur numérique Q. Nous définissons le niveau de charge d'un nœud X (**NvCh(X)**) comme la partie inférieure entière du rapport de sa charge locale Q sur une valeur N.

$$\text{Soit}: NvCh(X) = \lfloor Q/N \rfloor$$

où N est la quantité de charge par niveau.

Dans notre cas particulier la charge locale d'un noeud est quantifiée par le nombre de processus sur ce noeud, soit en état d'exécution, soit en état d'attente de communications. Par exemple si N = 5 processus, le niveau de charge 1 correspond à $Q = 5$ à 9 processus, 2 correspond à $Q = 10$ à 14, etc.
Il est possible de prendre en compte d'autres critères pour quantifier la charge locale d'un noeud (la longueur de la file d'entrées/sorties ou de la file de traitement par

exemple), en utilisant la même formule pour le calcul du niveau de charge du noeud.

Pour chaque nœud X la connaissance partielle de l'état de charge de la machine est vue comme un sous-ensemble d'autres nœuds avec lesquels il peut partager sa charge. Ce sous-ensemble de nœuds est appelé la **Solution de Placement (SP)** du nœud X.

En disposant d'un groupe de nœuds considerés comme les plus appropriés pour le partage de charge, on diminue la fréquence de recherches de sites de placement et donc le temps d'exécution d'une application.

Les nœuds qui font partie d'une SP ont tous le même niveau de charge et le niveau de charge d'une SP est donc celui des noeuds qui la composent. La taille d'une Solution de Placement est variable et elle peut être supérieure ou égale à zero.

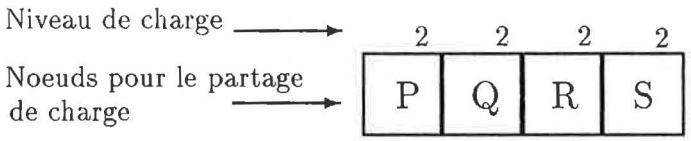


FIG. 4.1 - *Une Solution de Placement*

Sur la figure 4.1 nous avons une représentation graphique d'une SP du noeud X, composée par les noeuds P, Q, R et S. Sur cet exemple le niveau de charge de la SP est 2 et sa taille est 4.

Si on considère un système parallèle homogène (par exemple une grille) qui exécute une seule application, nous pouvons dire que les meilleures Solutions initiales de Placement pour chaque nœud peuvent être définies comme étant les nœuds voisins directement connectés (charge initiale nulle avant placement). Un nœud peut donc appartenir à différentes Solutions de Placement. Sur la figure 4.2 nous pouvons voir la représentation d'une grille de taille 5x5 et les SP initiales des nœuds 8 et 14.

Un noeud pourra partager sa charge équitablement entre les sites compris dans sa Solution de Placement. Lorsque les sites seront considérés comme étant sur-chargés, un nouvel ensemble de Solutions de Placement devra être construit.

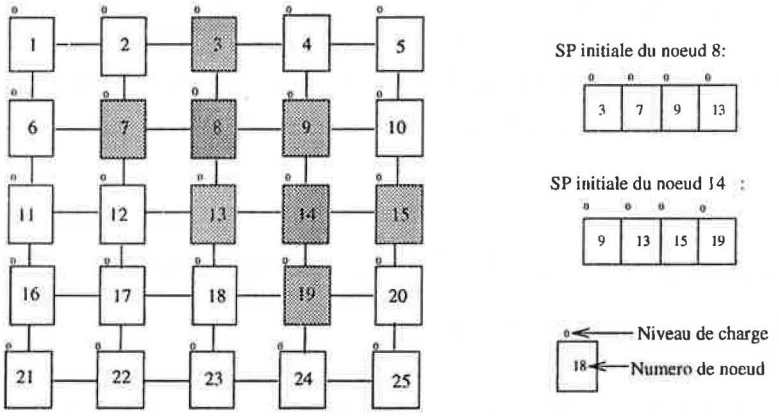


FIG. 4.2 - *La connaissance partielle initiale de l'état du système*

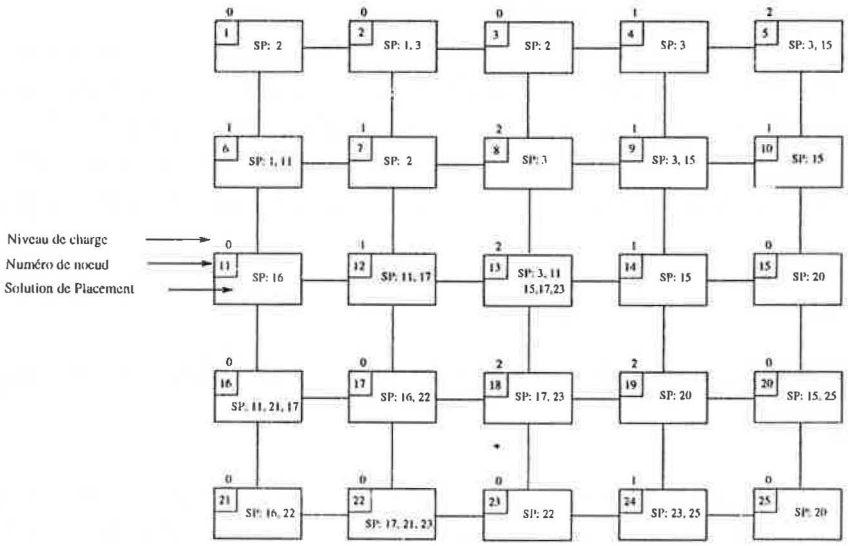


FIG. 4.3 - *Les SP des noeuds dans un état donnée du système*

La SP d'un nœud est une connaissance qui va évoluer pour s'adapter aux changements de la charge du système. A partir d'un état initial de charge et des SP, comme le montre la grille de la figure 4.2, lorsqu'une certaine application est exécutée, nous pouvons arriver à avoir, par exemple, un autre état où les noeuds possèdent différents niveaux de charge et d'autres SP comme indiqué sur la figure 4.3. Les SP des noeuds évoluent de façon à contenir quelques uns des sites les moins chargés et c'est en appliquant les opérateurs expliqués dans la section suivante que cela sera possible.

4.3 Actualisation de la connaissance partielle de l'état de la machine

Lorsque l'état du système change, les Solutions de Placement des nœuds vont évoluer sous l'effet de l'occurrence des événements suivants:

- Un noeud dans une SP dépasse le niveau de charge de la SP
- Un noeud dans une SP refuse d'accepter les processus d'autres nœuds
- Un noeud dans une SP acquiert un niveau de charge plus petit que celui de la SP.

Les opérateurs qui actualisent une Solution de Placement sont appliqués sur chaque noeud par le module d'allocation, après avoir reçu des messages des voisins directs lui annonçant leur augmentation ou leur diminution de niveau de charge. Ces événements provoquent l'ajout ou la suppression des noeuds dans les SP ainsi que la reconstruction de nouvelles SP. Dans la suite nous expliquons plus en détail ces opérateurs.

4.3.1 Opérateurs de croissance, de déclin et de remplacement

La SP d'un nœud est une connaissance qui va évoluer pour s'adapter aux changements de la charge du système. L'évolution des SP est déterminée par l'application des opérateurs suivants:

- Le déclin :

Lorsqu'un nœud X reçoit un message d'un voisin direct Y lui annonçant son augmentation de niveau de charge, si le nœud Y appartient à la Solution de Placement SP_x du nœud X et si

$$NvCh(nœudY) > NvCh(SP_x)$$

alors le nœud X appliquera l'opérateur de déclin sur SP_x pour enlever le nœud Y. La taille de SP_x sera donc réduite d'un après cette suppression.
L'opérateur de déclin s'applique aussi lorsque un nœud Y dans la SP refuse d'accepter la charge d'autres processus. Dans ce cas nous supposons que le noeud Y a probablement augmenté son niveau de charge et alors nous décidons de l'enlever de la SP_x.

- La croissance :

Cet opérateur s'applique lorsqu'un nœud X reçoit un message d'un nœud voisin direct Y lui annonçant sa diminution de niveau de charge. Si le nœud Y n'appartient pas à la SP du nœud X (SP_x) et si

$$NvCh(nœudY) \text{ est égal au } NvCh(SP_x)$$

alors le nœud X appliquera l'opérateur de croissance sur SP_x pour ajouter le nœud Y. La taille de SP_x sera donc augmentée d'un après cette opération.

- Le remplacement :

L'opération de remplacement a comme paramètres deux Solutions de Placement SP_i,SP_j avec NvCh(SP_i) $\neq$ NvCh(SP_j). Le résultat du remplacement est:

$$remplacement(SP_i, SP_j) = \begin{cases} SP_i & \text{si } NvCh(SP_i) < NvCh(SP_j) \\ SP_j & \text{si } NvCh(SP_i) > NvCh(SP_j) \end{cases}$$

Lorsqu'un nœud X avec une Solution de Placement SP_{x1} reçoit un message d'un voisin direct Y lui annonçant sa diminution de niveau de charge, et si le nœud Y remplit la condition

$$NvCh(nœudY) < NvCh(SP_{x1})$$

alors SP_{x1} sera remplacée par la Solution de Placement SP_{x2} formée par le nœud Y. Ce type de remplacement concerne la substitution d'une SP par une autre SP de taille un. L'opération de remplacement sera aussi utilisée par l'opérateur de fusion dont nous allons parler plus tard.

Un exemple du fonctionnement de ces opérateurs est le suivant:

En reprenant la figure 4.2, supposons que les noeuds 3 et 9 augmentent leur niveau de charge. Ils envoient donc un message à leurs voisins pour leur annoncer cette augmentation. En particulier si l'on s'intéresse aux actions effectuées par les nœuds 8 et 14 suite à cet événement, le nœud 8 va appliquer deux fois l'opérateur de déclin pour enlever les nœuds 3 et 9 et le nœud 14 appliquera le déclin pour supprimer le nœud 9 (Figure 4.4).

Si le niveau de charge du nœud 9 à nouveau diminue, il enverra un message à ses voisins pour leur annoncer cette diminution. En regardant les actions effectuées par les nœuds 8 et 14 et puisque le niveau de charge du nœud 9 est égal au niveau de charge de leurs Solutions de Placement, ils vont appliquer l'opérateur de croissance pour le rajouter. Les tailles des SP des nœuds 8 et 14 seront 3 et 4 respectivement (voir figure 4.5).

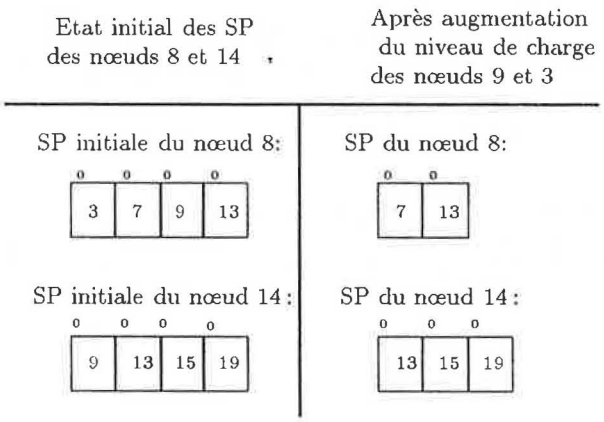


FIG. 4.4 - *Exemple de fonctionnement de l'opérateur de déclin*

Supposons qu'on se trouve dans une situation spéciale où le nœud 8 a une SP avec un niveau de charge égal à 1 et les nœuds 2,4,12,14 en font partie. Si le noeud 8 reçoit un message du nœud 9 lui annonçant la diminution de son niveau de charge avec une nouvelle valeur égale à zéro, il remplacera sa SP par une autre de taille 1 formée par le nœud 9 (figure 4.6). L'utilisation de l'opérateur de remplacement sera aussi illustrée pendant l'explication de l'opérateur de fusion.

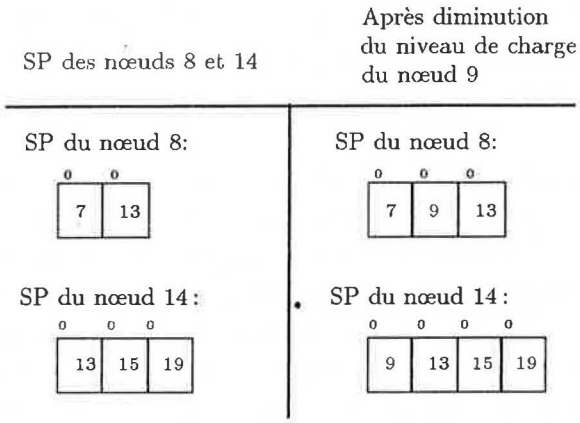


FIG. 4.5 - *Exemple de fonctionnement de l'opérateur de croisement*

4.3.2 L'Opérateur de fusion

Après plusieurs applications de l'opérateur de déclin, on peut arriver à l'obtention d'une Solution de Placement vide.

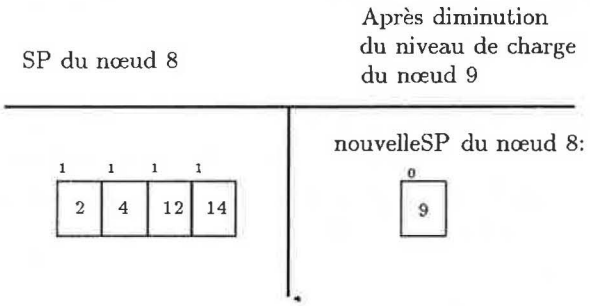


FIG. 4.6 - *Exemple de remplacement d'une SP par une autre de taille 1*

Lorsque la SP d'un nœud est vide, ce nœud envoie des requêtes aux nœuds voisins en leur demandant une copie de leurs SP. Le but est de reconstruire une nouvelle SP qui permettra d'élargir le domaine initial de placement. Cette reconstruction se fait en utilisant l'opérateur de fusion.

L'opérateur de fusion s'applique sur un ensemble de SP qui peuvent avoir différentes tailles et différents niveaux de charge. La fusion peut se faire de deux manières:

- **Fusion des voisins.** Sur N Solutions de Placement de taille 1 dont le seul composant est un nœud voisin directement connecté (N est égal au nombre de voisins).

- **Fusion des SP des voisins.** Sur M Solutions de Placement provenant des nœuds voisins. Les tailles de ces Solutions de Placement peuvent être supérieures ou égales à 1.

La première s'utilise lorsqu'aucune SP demandée aux nœuds voisins par le nœud X n'est pas encore arrivée et que le nœud X a besoin de placer un processus. Le nœud interprétera cet événement de la même façon que si tout le système avait dépassé un premier niveau de charge et que les Solutions de Placement des autres noeuds sont vides. Le nœud X prend alors la décision de tout recommencer et de reconstruire une Solution de Placement qui comprend les nœuds voisins mais avec un niveau de charge supérieur au niveau initial. Pour réaliser cette opération, chaque nœud X dans la machine possède un tableau contenant une copie des niveaux de charge de ses voisins directement connectés qui ont formé sa solution initiale de placement. Ce tableau est mis à jour durant l'exécution des applications chaque fois que les noeuds voisins annoncent leurs augmentations ou leurs diminutions de niveau de charge.

Dans le deuxième cas, les Solutions de Placement des nœuds voisins commencent à arriver. Pour ne pas bloquer le déroulement de l'application parallèle, l'opérateur de fusion est appliqué de façon asynchrone sur chaque paire de Solutions de Placement au fur et à mesure qu'elles arrivent.

Le choix de l'opérateur de fusion a une grande influence sur l'équilibrage de la charge dans la machine parallèle. Nous avons défini un type de fusion qui construit une SP à partir de k solutions de placement en prenant seulement les nœuds qui appartiennent aux m $\leq$ k Solutions de Placement avec le même niveau de charge minimal.

Soit SP_i la SP du noeud i et SP_j une SP non vide du noeud j (voisin directement connecté au noeud i). D'une manière générale et en prenant les SP comme des ensembles d'éléments, le fonctionnement de l'opérateur de fusion pour construire une nouvelle SP_i est décrit de la façon suivante:

Pour toute paire de Solutions de Placement SP_i, SP_j:
Si (non_vide(SP_i))
{
    Si NvCh(SP_i) $\neq$ NvCh(SP_j)
        fusion(SP_i,SP_j) = remplacement(SP_i,SP_j)
    sinon
        fusion(SP_i,SP_j) = $SP_i \cup SP_j$
}
sinon
    fusion(SP_i,SP_j) = SP_j

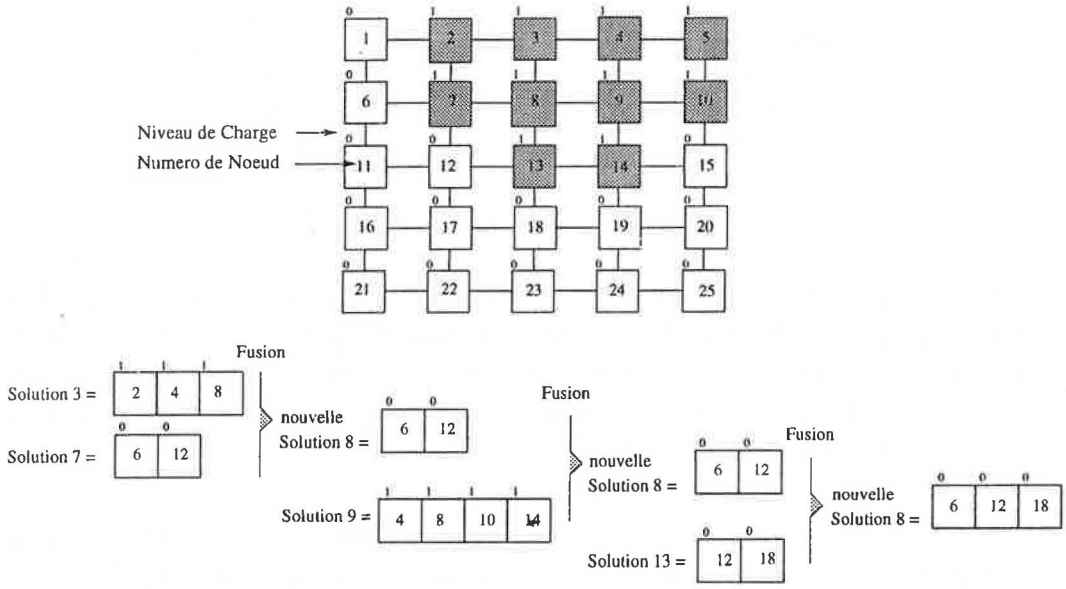


FIG. 4.7 - *L'opérateur de fusion pour la construction d'une SP du nœud 8*

La figure 4.7 montre un exemple de cet opérateur sur une configuration donnée de l'état de charge de la machine; les nœuds ayant un niveau de charge plus grand que zéro sont grisés. Dans cet exemple la Solution de Placement SP_8 du nœud 8 est vide; après avoir envoyé des requêtes de SP à ses voisins (les nœuds 3,7,9 et 13) supposons que l'ordre d'arrivée des SP est SP_3, SP_7, SP_9 *et* SP_{13}. Puisque SP_8 est vide, elle prendra d'abord la valeur de SP_3. Ensuite le nœud 8 appliquera l'opérateur de fusion sur chaque paire de SP et à la fin obtiendra une nouvelle SP_8 formée par les nœuds qui font partie des SP avec le même niveau de charge minimal (SP_7 et SP_{13}). Nous faisons remarquer que les différentes SP_8 obtenues en cours de la reconstruction peuvent être utilisées pour le placement de processus sans avoir besoin d'attendre l'arrivée de toutes les SP des voisins.

Nous avons présenté les opérateurs qui rendent possible l'évolution de la connaissance sur l'état de charge de la machine, dans le paragraphe suivant nous expliquons l'opérateur de rotation utilisé, non pas pour faire évoluer les SP, mais pour obtenir une meilleure répartition de la charge entre les nœuds faisant partie d'une SP.

4.3.3 L'opérateur de rotation

L'ensemble de nœuds qui forment une SP peuvent être vus comme une liste. Pour accéder à tous les nœuds et donc partager la charge équitablement, nous appliquons l'opérateur de rotation pour passer le nœud qui se trouve dans la première position de la liste à la dernière position (figure 4.8).

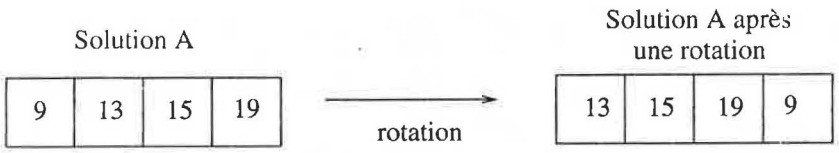


FIG. 4.8 - *Opérateur de rotation*

Nous avons défini des opérateurs qui manipulent des SP avec le même niveau de charge pour tous les nœuds qu'elles contiennent. Nous appelons ce type de SP une Solution de Placement homogène par rapport au niveau de charge. C'est le changement de niveau qui rend possible l'application des opérateurs de déclin, de croissement, de fusion et de remplacement.

Nous soulignons que d'autres types de SP non homogènes peuvent être construits et que des nouvelles conditions d'application des opérateurs peuvent donc être définies. Par exemple un autre type de SP est défini comme un ensemble d'au maximum N nœuds ordonnés de façon ascendante par rapport à leurs niveau de charge (où N est un entier ≥ 0). Les opérateurs de croissance et de déclin pourraient, par exemple, réordonner les noeuds dans la SP lorsque leurs niveaux de charge changent. Dans le cas où un noeud dans la SP n'accepte plus une charge externe, il sera éliminé de la SP. L'opérateur de fusion dans cet exemple pourrait être défini de telle façon qu'il construise des SP avec les nœuds qui ont les niveaux de charge les plus petits ordonnés de manière ascendante. La fusion de deux SP doit générer une nouvelle SP dont la taille est inférieure ou égale à N. Ce deuxième type de SP a été implanté et comparé avec la SP homogène, nos résultats ont montré (section 5.2.4) que les SP homogènes donnent de meilleures performances.

Dans la section suivante nous présentons les algorithmes qui définissent l'élément de contrôle de cette méthode évolutive pour équilibrer la charge. Ces algorithmes utilisent les Solutions de Placement pour décider quand et où placer les processus qui se créent pendant l'exécution.

4.4 L'Algorithme Evolutif

Dans l'Algorithme Evolutif, l'élément de contrôle est distribué. Comme il a été dit précédemment cet élément englobe les deux politiques de transfert et de localisation pour déterminer, quand et sur quel nœud placer un processus.

La politique de transfert de l'algorithme évolutif suivie par chaque nœud X, dépend

de la comparaison du niveau de charge du nœud X et du niveau de charge de sa Solution de Placement SP_X pour décider si un processus sera exécuté localement ou sur un autre noeud. Quant à la politique de localisation, le choix d'un site d'exécution pour un processus P est simple; pour déterminer sur quel nœud un processus sera placé, un nœud X prendra directement un nœud Y $\in SP_X$ et lui enverra une requête d'acceptation du processus P.

L'algorithme d'équilibrage que nous présentons a été implanté sous un comportement de placement définitif, c'est-à-dire qu'une fois qu'un nœud a accepté une requête, il s'engage à recevoir et à exécuter le processus sans le déplacer à nouveau; pour cela nous utilisons un mécanisme d'acceptation/refus de requêtes.
Cependant l'algorithme évolutif peut être implanté sous un schéma supportant la migration de processus; il faudrait dans ce cas faire le choix du processus à migrer et modifier le moment d'activation du processus qui décide des transferts (fait actuellement au moment de la création des processus); l'activation pourrait être faite soit sous une fréquence donnée, soit après l'occurrence d'un certain événement.

L'Algorithme Evolutif est composée de plusieurs actions qui s'exécutent d'une manière concurrente sur chaque nœud de la machine lorsque certains événements arrivent. La figure 4.9 montre les trois blocs qui regroupent d'une manière générale ces actions. Le premier bloc comprend les actions ayant un rapport avec le placement des processus de l'application. Le deuxième bloc regroupe les actions associées à l'échange des messages annonçant les changements de niveau de charge des nœuds, et finalement le troisième bloc comprend les actions de l'algorithme évolutif associées à la manipulation des Solutions de Placement.

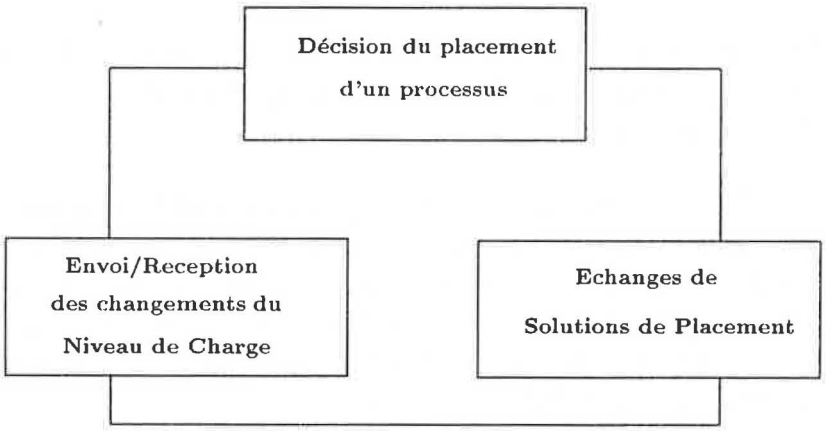


FIG. 4.9 - *Schéma global des actions qui composent l'algorithme Evolutif*

Les échanges d'information entre ces blocs sont montrés sur la figure 4.10. Les événements qui constituent l'Algorithme Evolutif ont une représentation graphique en

forme d'ovale. Les principales actions possibles associées aux événements sont entourées par des rectangles.

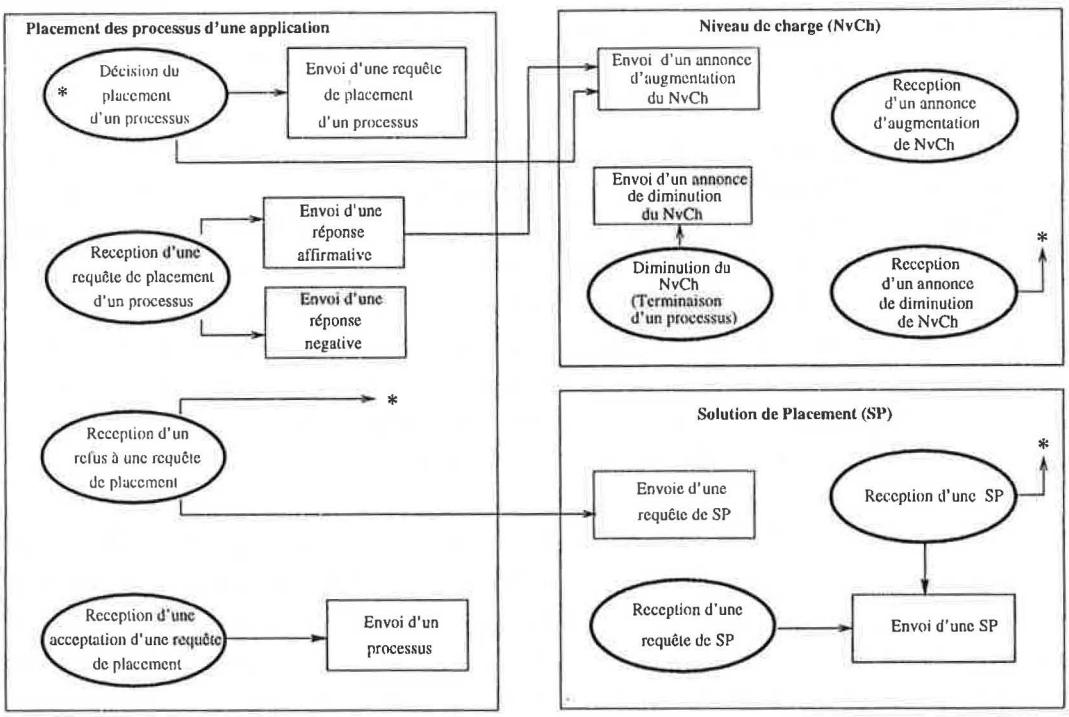


FIG. 4.10 - *Les événements et les échanges d'information de l'Algorithme Evolutif*

Le premier bloc contient quatre événements concernant le placement des processus d'une application. En suivant une politique d'acceptation et de refus les actions effectuées sont les suivantes:

- Lorsque l'événement concerne la création par un noeud X d'un processus d'une application, deux actions possibles peuvent être exécutées. Soit le noeud X ne peut pas exécuter le processus et donc il envoie une requête de placement du processus vers les nœuds dans la SP_X. Soit le noeud X exécute le processus et si son niveau de charge augmente, il envoie un message à ses voisins directement connectés pour annoncer l'augmentation.

- Si l'événement est une réception par un noeud X d'une requête de placement d'un processus, l'action à exécuter est d'analyser la possibilité de recevoir ce processus et de répondre à la requête de manière affirmative ou bien avec un refus. Si le noeud X a accepté le processus, il le prend en compte pour calculer

son nouveau niveau de charge et si celui-ci augmente, le noeud X envoie un message à ses voisins directement connectés pour annoncer l'augmentation.

- Lorsque l'événement indique la réception par un noeud X d'un refus à une requête de placement de la part du noeud Y, l'action à suivre est d'appliquer l'opérateur de déclin sur la SP_X pour enlever le noeud Y. Si SP_X devient vide le noeud X envoie aux voisins directement connectés une requête de SP. Ensuite, le noeud X essaye de placer le processus refusé et fait la même action que s'il s'agissait de l'occurrence d'un événement pour la création d'un processus.

- Si l'événement indique la réception par un noeud X d'une acceptation à une requête de placement de la part du noeud Y, l'action à suivre est d'envoyer le processus au noeud Y.

Le deuxième bloc est formé par trois événements concernant les changements des niveaux de charge qui déclenchent certaines actions pour l'application des opérateurs évolutifs de l'algorithme:

- La réception par un noeud X d'un message provenant d'un voisin direct Y annonçant son augmentation de niveau de charge, peut provoquer l'application de l'opérateur de déclin du voisin Y sur SP_X.

- La réception par un noeud X d'un messages provenant d'un voisin direct Y annonçant sa diminution de niveau de charge, peut provoquer l'application des opérateurs de remplacement ou de croissance du voisin Y sur SP_X. Dans le cas où le noeud X a des processus dans une file d'attente d'un site de placement, il essaye de les placer en faisant la même action que s'il s'agissait de l'occurrence d'un événement pour la création d'un processus.

- L'action à suivre par le noeud X lorsqu'il y a une terminaison d'un processus du programme est de recalculer son niveau de charge. Si celui-ci a changé le noeud X envoie un message à ses voisins directs annonçant le changement. Dans notre implantation, puisque la charge d'un nœud est calculée par le nombre de processus sur le nœud, la détection d'une diminution de niveau de charge se fait au moment de la terminaison de l'exécution des processus.

Deux événements activent les actions qui font partie du dernier bloc du schéma 4.10 concernant l'évolution des Solutions de Placement des nœuds:

- La réception par un noeud X d'une requête de Solution de Placement provenant d'un voisin direct Y, implique l'envoi de SP_X vers le voisin Y si SP_X n'est

pas vide. Si SP_X est vide, le voisin Y sera mis dans la file d'attente de SP du noeud X.

- La réception par le noeud X d'une SP provenant d'un voisin direct Y provoque l'application de l'opérateur de fusion de SP_Y et de SP_X. Dans le cas où le noeud X a des processus dans une file d'attente d'un site de placement, il essaye de les placer en faisant la même action que s'il s'agissait de l'occurrence d'un événement pour la création d'un processus. Si la file d'attente de SP du noeud X n'est pas vide, le noeud X envoie sa nouvelle SP_X aux noeuds dans cette file.

Dans la suite une explication plus détaillée des actions effectuées dans l'Algorithme Evolutif est présentée.

4.4.1 L'action qui se déclenche au moment de la création des processus d'une application

L'action de base qui est exécutée sur tout nœud X dans la machine, lorsqu'un processus de l'application doit être créé sur ce noeud, décide du transfert ou bien de l'exécution locale du processus en se basant sur le rapport des niveaux de charge du nœud X et de sa Solution de Placement SP_X. Si NvCh(nœud X) $\leq$ NvCh(SP_X) alors les processus de l'application seront placés sur le nœud X, autrement on cherchera à partager la charge en les transférant vers les nœuds appartenant à la SP_X. L'algorithme suivant décrit le fonctionnement de cette activité.

```
PROC Décision de placement prise par un nœud X
{
    SI (vide(SP_X)) SP_X = Fusion des voisins;
    SI ( ( NvCh(Noeud X) ≤ NvCh(SP_X) ) ET
      ( NvCh(nœud X) ≠ SATURE ) )
    {
          placer le processus sur le noeud X;
          SI ( NvCh(nœud X) a augmenté )
               Annoncer aux noeuds voisins l'augmentation;
    }
    SINON
    {
          SI (( non vide (SP_X)) ET
```

```
            ( NvCh(SP_X) ≠ SATURE) )
        {
            choisir un nœud Y ∈ SP_X;
            envoyer une requête de placement au nœud Y;
            Rotation(SP_X);
        }
        SINON
            insérer le processus dans la file d'attente de sites;
    }
}
```

Si une SP est vide (les SP des voisins ne sont pas encore arrivées) et que le noeud X doit prendre une décision de placement, on reconstruit une nouvelle SP provisoire pour le noeud X. La nouvelle SP est faite en prenant les voisins directement connectés, qui ont le même niveau de charge minimal (comme la SP initiale mais avec un niveau de charge plus grand). Cette action permettra de ne pas placer trop loin les processus créés et, en conséquence, de ne pas surcharger les liens de communication. Lorsque les SP des voisins arriveront, l'opérateur de fusion sera appliqué et la SP du noeud X sera à nouveau modifié.

D'autre part, on peut voir que si le processus de l'application a été placé localement, le niveau de charge du noeud X peut changer. L'envoi du nouveau niveau de charge aux voisins est fait seulement lorsqu'il y a eu une augmentation.

Dans l'implantation qui a été faite, on a choisi le premier nœud Y dans SP_X pour lui envoyer une requête de placement et on applique ensuite l'opérateur de rotation sur SP_X pour partager équitablement les processus entre tous les nœuds qu'elle contient. Cependant ce critère peut être modifié en transférant tous les processus au même nœud jusqu'au changement de son niveau de charge.
Si le nœud X et la SP_X ont un niveau de charge considéré comme saturé, les processus seront mis dans une file d'attente et seront placés quand d'autres Solutions de Placement arriveront. Dans nos expériences le niveau de charge SATURE est un numéro qui représente un nombre maximal de niveaux de charge sur chaque nœud.

A côté de cette action de base il existe d'autres actions qui déclenchent directement ou indirectement l'actualisation de la connaissance après l'application des opérateurs évolutifs. Ces actions s'activent lorsqu'arrive soit une requête de placement, soit le refus ou l'acceptation d'une requête, soit le changement de la charge des noeuds voisins, soit les SP demandées aux voisins (voir section 4.3). Dans la suite de ce document, ces actions sont présentées.

4.4.2 L'action exécutée lors de l'arrivée d'une requête de placement

Cette action s'active sur un nœud Y lorsqu'il arrive un message provenant du nœud X qui contient une requête de placement d'un processus P. Pour répondre, le nœud Y prend en compte son niveau de charge par rapport à celui de sa solution de placement SP_Y et suit l'algorithme ci-dessous :

```
PROC  Répond requête
{
    SI (( NvCh(nœud Y) <= NvCh(SP_Y) ) ET
            (NvCh(nœud Y) ≠ SATURE) )
    {
            envoie un message d'ACCEPTATION au nœud X;
            réserver la mémoire pour le processus P;
            SI (NvCh(nœud Y) a augmenté)
                Annoncer aux noeuds voisins l'augmentation;
    }
    SINON
            envoie un message de REFUS au nœud X
}
```

Une fois qu'un nœud Y a accepté la requête de placement d'un processus P, il réserve l'espace de mémoire nécessaire pour exécuter le processus P et il ne pourra pas le transférer à un autre noeud. Puisque la mesure de la charge dépend du nombre de processus par processeur, si un nœud s'engage à recevoir un processus externe, son niveau de charge peut être modifié, même si le processus ne lui a pas encore été transmis. Lorsqu'il y a une augmentation de niveau de charge, le nœud Y l'annonce à ses voisins.

4.4.3 Les actions effectuées lors de la reception d'une réponse à une requête de placement

Lorsqu'un message arrive au nœud X indiquant la réponse d'une requête de placement, il peut y avoir deux actions à suivre. Dans le premier cas, s'il s'agit d'une

réponse affirmative provenant du nœud Y qui a accepté d'exécuter le processus P, le nœud X enverra le processus P au nœud Y.

Dans le cas contraire, si le nœud Y refuse la requête du nœud X, le nœud X appliquera l'opérateur de déclin pour enlever le nœud Y de sa solution de placement et choisira un autre nœud Z $\in SP_X$ pour faire une nouvelle requête de placement. Dans l'algorithme suivant nous montrons le comportement de cette action.

```
PROC requête refusée
{
    Déclin(SP_X,nœud Y);
    SI ( vide(SP_X) )
    {
        SI ( nb_de_SP_attendues == 0 )
        {
            envoyer aux voisins un message de requête de SP;
            nb_de_SP_attendues = nb de voisins;
        }
        SP_X = Fusion des voisins;
    }
    SI ( ( NvCh(nœud X) <= NvCh(SP_X) ) ET
       ( NvCh(nœud X) ≠ SATURE ) )
    {
        placer le processus sur le nœud X;
        SI (NvCh(nœud X) a augmenté)
            Annoncer aux nœuds voisins l'augmentation;
    }
    SINON
    {
        SI ( NvCh(SP_X) ≠ SATURE )
        {
            choisir un nœud Y ∈ SP_X;
            envoyer une requête de placement au nœud Y;
            Rotation(SP_X);
        }
        SINON
            inserer le processus dans la queue d'attente de sites;
    }
}
```

C'est ici que la SP peut rester vide après plusieurs utilisations de l'opérateur de déclin sur SP_X pour enlever les nœuds qui n'acceptent plus de processus extérieurs. Lorsque SP_X reste vide et si on ne l'a pas déjà fait, le nœud X enverra des requêtes de SP aux voisins directs dans le but de pouvoir reconstruire ultérieurement une nouvelle SP_X. La variable nb_de_SP_attendues indique le nombre de SP attendues initialement, si sa valeur est nulle cela nous indique qu'il n'y a pas eu de requêtes de SP déjà faites.

Si SP_X reste vide, pour ne pas dépendre de l'arrivée des SP des voisins et pour prendre en compte une possible diminution du niveau de charge local, le nœud X reconstruit ensuite une nouvelle SP_X en appliquant l'opérateur de fusion sur ses voisins. Dans le cas où le nœud X a diminué son niveau de charge par rapport à ses voisins, il va placer le processus sur lui même, et dans d'autre cas, il enverra une requête de placement à un voisin.

4.4.4 L'action exécutée lors de la reception d'un message qui annonce la diminution du niveau de charge d'un voisin direct

Lorsque le nœud X reçoit un message annonçant la diminution du niveau de charge du nœud Y, il peut y avoir deux cas, le premier cas se présente quand SP_X n'est pas vide et le nœud Y appartient à SP_X, l'action à suivre est de comparer le NvCh(nœud Y) avec le niveau de charge de SP_X pour appliquer l'opérateur de remplacement ou bien de croissance en exécutant l'algorithme suivant:

```
PROC  Charge diminuée du nœud Y
{
    mettre à jour le tableau des niveaux de charge des voisins(nœud Y);
    SI ( non vide(SP_X) )
    {
            SI ( NvCh(SP_X) > NvCh(nœud Y) )
            Remplacer(SP_X,nœud Y);
            SINON
            {
            SI ( NvCh(SP_X) == NvCh(nœud Y) )
                croissance(SP_X,nœud Y);
    }
    SINON
    {
            Remplacer(SP_X,nœud Y);
```

```
            SI ( non vide(liste_de_processus_en_attente) )
                décision du placement des processus en attente;
            SI ( non vide(liste_de_requêtes_de_SP) )
                envoyer SP_X aux nœuds dans la liste_de_requêtes_de_SP;
    }
}
```

On peut avoir un deuxième cas où SP_X est vide, il existe la possibilité d'avoir des processus en attente d'un site de placement. Pour trouver une solution à ce problème, le nœud Y deviendra la nouvelle SP_X avec un seul élément et le nœud X enverra des requêtes de placement au nœud Y comme dans la création des processus.

4.4.5 L'action effectuée lors de l'arrivée d'un message qui annonce l'augmentation du niveau de charge d'un voisin direct

Lorsqu'un nœud X reçoit un message d'annonce d'augmentation du niveau de charge du nœud Y, il met à jour le tableau des niveaux de charge de ses voisins directs. Ensuite le nœud X peut appliquer l'opérateur de déclin du nœud Y sur SP_X et s'il est necéssaire, envoyer des requêtes de SP aux voisins. Un tel comportement est représenté par l'algorithme suivant:

```
PROC Arrive chargeaugmentée du nœud Y
{
    mettre à jour le tableau des niveaux de charge des voisins(nœud Y);
    SI ( non vide(SP_X) )
    {
        SI ( ( nœud Y ∈ SP_X) et (NvCh(SP_X) < NvCh(nœud Y) ) )
        Declin(SP_X,nœud Y);
        SI (vide(SP_X) et (nb_de_SP_attendues == 0) )
        {
            envoyer les requêtes des SP aux voisins;
            nb_de_SP_attendues = nb de voisins;
        }
    }
}
```

Nous utilisons la variable nb_de_SP_attendues pour indiquer combien de SP on attend et si cette variable est égale à zero cela indique que l'on ne peut plus attendre

de SP et qu'il faut donc lancer des nouvelles requêtes. Le nombre de SP attendues initialement est égal au nombre des voisins directement connectés au nœud X.

4.4.6 L'action qui se déclenche lors de la terminaison de l'exécution d'un processus de l'application sur un noeud

A chaque fois qu'un processus termine son exécution sur un nœud X, il y a une re-évaluation du niveau de charge et si le nouveau NvCh(nœud X) a diminué par rapport au dernier, le nœud X annonce à ses voisins directs sa diminution de niveau de charge. Ce comportement est décrit dans l'algorithme suivant:

PROC *Diminution de Charge du nœud X*
{
calculer nouveau Niveau de Charge $NvCh_{X2}$;
SI ($NvCh_{X2} < NvCh_X$)
annoncer aux voisins la diminution du niveau de charge($NvCh_{X2}$);
$NvCh_X = NvCh_{X2}$;
}

4.4.7 L'action effectuée lors de la reception d'un message contenant une Solution de Placement envoyée par un voisin direct

Lorsqu'un nœud X reçoit les SP demandées à ses voisins, il applique l'opérateur de fusion pour en reconstruire une nouvelle. C'est ici qu'on décrémente la variable nb_de_SP_attendues comme l'indique l'algorithme suivant

PROC Arrivesolutionde placement SP_Y d'un nœud voisin Y
{
dernière_SP_arrivée = 0;
SI (nb_de_SP_attendues > 0)
{
nb_de_SP_attendues = nb_de_SP_attendues - 1;
SI (nb_de_SP_attendues == 0) dernière_SP_arrivée = 1;
}
SP_X = fusion(SP_X,SP_Y);
SI (non vide(file_de_requêtes_de_SP)
envoyer SP_X aux nœuds dans la liste_de_requêtes_de_SP;

```
    SI ( derniere_SP_arrivée == 1 )
            vider(file_de_requêtes_de_SP);
    SI ( liste processus en attente)
            .       décision du placement des processus en attente;
}
```

Si la file de requêtes de SP du nœud X ($FRSP_X$) n'est pas vide, le nœud X envoie sa nouvelle SP_X aux nœuds $\in FRSP_X$, même s'il attend d'autres SP. Une fois que le nœud X a reçu la dernière SP attendue, il vide sa $FRSP_X$. Cela veut dire que seulement les K premières SP arrivées pourront être prises en compte pour être envoyées aux nœuds $\in FRSP_X$ (K = nb de voisins). Comme dans le cas du processus qui reçoit les messages qui annoncent les diminutions des niveaux de charge, le nœud X devra utiliser sa nouvelle SP_X pour essayer de placer les processus qui sont dans la file d'attente des sites.

4.4.8 L'action déclenchée lors de la reception d'une requête de Solution de Placement

Dans l'algorithme suivant, nous pouvons voir comment un nœud X réagit lorsqu'il reçoit une requête de SP provenant du nœud Y. Dans le cas où SP_X est vide le nœud X mettra le nœud Y dans la file d'attente de SP, mais si SP_X n'est pas vide alors il faudra prendre en compte la dernière date où l'on a appliqué l'opérateur de fusion pour la mettre à jour

```
PROC  Arrive reqêtede SP
{
    SI ( non vide(SP_X) )
    {
            SI ((date_actuelle - dernière_date_requête_SP) ≥ TEMPS_REQ_SP )
            {
                envoyer aux voisins un message de requête de SP;
                nb_de_SP_attendues = nb de voisins;
                ajoute_nœud(file_de_requêtes_de_SP,nœud Y);
                rendre vide SP_X;
            }
            SINON
                SI ( NvCh(SP_X) ≠ SATURE )
                    envoyer SP_X au nœud Y;
    }
    SINON
```

```
            ajoute_nœud(liste_demandes_SP,nœud Y);
}
```

Puisque l'actualisation des SP se déclenche lorsqu'il y a des créations de processus (principalement pour provoquer l'application des opérateurs de déclin et de fusion), si un nœud Y ne fait pas de créations il risque de bloquer l'évolution de la connaissance sur l'état de la machine. Pour résoudre ce problème on a fixé un intervalle maximal de temps pour lequel une SP est valable. Après cette spécification, si un nœud X a une SP_X non vide et s'il reçoit une requête de SP provenant du nœud Y, d'abord il vérifie le temps de validité de sa SP_X en faisant une comparaison entre la date actuelle et la dernière date à laquelle il a envoyé une requête de SP. Si la SP_X est toujours valable le nœud X l'enverra au nœud Y, dans le cas contraire, le nœud X mettra le nœud Y dans la file d'attente de SP et fera une requête de SP à ses voisins pour obtenir une nouvelle SP_X qui sera envoyée au nœud Y postérieurement.

4.5 Conclusion du chapitre

En guise de conclusion, nous donnons un aperçu global du comportement de l'algorithme évolutif pour l'équilibrage dynamique de la charge. L'algorithme présenté dans ce chapitre est caractérisé par l'introduction de plusieurs opérateurs évolutifs qui sont appliqués pour actualiser la connaissance locale sur l'état de charge de la machine de chaque nœud . Cette connaissance locale d'un nœud X est représentée par un groupe de taille variable de nœuds, appelé la Solution de Placement du nœud X.

La décision de transfert d'un processus dépendra du rapport entre les niveaux de charge d'un nœud et de sa Solution de Placement; un nœud X essayera d'équilibrer sa charge par rapport à la charge des nœuds qui font partie de sa Solution de Placement.

L'introduction des niveaux de charge nous a permis de remplacer le besoin d'utiliser un système à double seuil pour équilibrer la charge. Un noeud dans la machine pourra augmenter son niveau de charge jusqu'au moment de devenir saturé. La capacité d'un niveau de charge (déterminée par un nombre de processus) a une influence directe sur la répartition des processus et sur la qualité du placement obtenu.

Dans l'algorithme évolutif l'efficacité de la politique de localisation pour choisir sur quel nœud un processus sera placé dépendra des conditions des liens de communica-

tions. Si un nœud reçoit rapidement d'autres SP pour en reconstruire une nouvelle, il pourra élargir son domaine de placement en utilisant plus de nœuds de la machine. Cependant lorsque les liens de communication sont chargés et qu'un nœud X ne reçoit aucune SP, pour ne pas bloquer l'exécution de l'application et pour ne pas saturer les liens de communication, le nœud X reconstruira une nouvelle SP en prenant en compte les nœuds voisins comme dans le cas initial mais avec un niveau de charge supérieur.

Comme nous avons expliqué, la reconstruction d'une Solution de Placement SP_X est provoquée lorsqu'il n'y a plus de nœuds dans SP_X. Il peut arriver que la charge de la machine change considérablement et qu'un nœud X doive vider sa SP pour mettre à jour sa connaissance, alors ce dernier n'as pas intérêt à construire des SP trop longues. Pour étudier le comportement de l'algorithme évolutif du point de vue de la longueur des SP, nous utilisons un paramètre qui contrôle la taille maximale des SP.

Un autre paramètre qui intervient dans la performance de l'algorithme évolutif est le temps maximal de validité d'une SP, cela dépend des caractéristiques de l'application par rapport au nombre de créations par processus. Dans le chapitre 5 nous allons montrer le comportement de cette méthode avec différentes valeurs des paramètres et présenter une évaluation expérimentale par simulation du fonctionnement de l'algorithme.

Chapitre 5

Simulation et études comparatives de l'Algorithme Evolutif

Sommaire

Les objectifs de ce chapitre sont l'étude du comportement de l'algorithme Evolutif et la présentation d'une comparaison avec d'autres algorithmes de placement dynamique pour l'exécution de certains jeux de test. Le simulateur des méthodes d'allocation dynamique SIMAD (présenté dans le chapitre 2) a été utilisé pour atteindre ces objectifs, effectuer les simulations et produire les résultats présentés.

Pour étudier le comportement de l'algorithme Evolutif et pour avoir une comparaison avec d'autres méthodes nous avons utilisé les programmes de test présentés dans la section 3.10.

5.1 Rappel du contexte d'étude et de simulation

Nous rappelons que ces programmes de test ont été divisés d'une manière générale en deux groupes:

- Le groupe 1 de programmes qui ne générent pas beaucoup de communications
  Ce groupe comprend des programmes formés par des processus qui pendant leur exécution communiquent peu. Dans ce cas la parallélisation des calculs est le plus importante. Les programmes de ce groupe sont aussi divisés en deux sous-groupes: les programmes avec un graphe de créations en forme d'arbre complet où chaque processus (qui n'est pas une feuille dans l'arbre) crée un nombre N de processus fils (le Groupe 1-1):

Num	Nom du programme	Annexe
1	processus-indep.prg	A.1
2	arbre-complet-1.prg	A.2
3	prg-all-1.prg	A.3
4	pparallel.prg	A.4
5	travaille-tjrs.prg	A.5
6	arbr1-AuDiAuDi.prg	A.6

  et les programmes avec d'autres graphes de créations généraux (le Groupe 1-2):

Num	Nom du programme	Annexe
7	p1pere-Nfils.prg	A.7
8	p1pere-Nfils-sync.prg	A.8
9	disques.prg	A.9
10	reines.prg	A.10
11	arbre-tableau.prg	A.11

- Le groupe 2 de programmes qui génèrent plusieurs échanges de messages
  Les programmes qui font partie du deuxième groupe sont constitués par des processus qui font plusieurs échanges de messages pendant leur exécution. Dans ce cas non seulement la parallélisation des calculs est importante, mais aussi les processus doivent échanger des messages et leur localisation sur la machine doit

être faite de façon adéquate pour ne pas surcharger les liens de communication. Comme dans le cas précédent, nous avons divisé les programmes de ce second groupe en deux sous-groupes: les programmes avec un graphe de créations en forme d'arbre complet (le groupe 2-1):

Num	Nom du programme	Annexe
12	arbre-complet-2.prg	B.1
13	prg-comm-1.prg	B.2
14	comm3FFH7.prg	B.3

et les programmes avec d'autres graphes de créations généraux (le groupe 2-2):

Num	Nom du programme	Annexe
15	espace-d-etats.prg	B.4
16	fourier.prg	B.5
17	fourier-par.prg	B.6
18	prodN-consM.prg	B.7

Dans la première partie de ce chapitre nous utilisons seulement certains programmes de test qui nous ont permis d'étudier le comportement de l'algorithme Evolutif. Nous faisons varier les paramètres qui sont à la base de son fonctionnement: la taille maximale des Solutions de Placement, le temps de validité des SP, le nombre de processus par niveau de charge et l'opérateur de fusion utilisé.

La deuxième partie du chapitre montre une évaluation de l'algorithme Evolutif basée sur la comparaison par simulation avec la méthode du Gradient (présentée dans la section 1.4) et une méthode aléatoire pour le placement dynamique de la charge. La comparaison a été obtenue en observant le comportement des méthodes pendant l'exécution des 2 groupes de programmes de test. Les critères principaux que nous avons choisis pour déterminer si un algorithme de placement dynamique est plus performant qu'un autre, prennent en compte les temps d'exécution des applications, le nombre de ressources utilisées et l'équilibrage de la charge qui sont des indicateurs pertinents pour notre étude.

La grille de processeurs a été utilisée dans le développement de nos plans d'expériences, car c'est un système dont la topologie nous a permis d'étudier et de comparer les algorithmes de placement dynamique: lorsque les noeuds possèdent une information partielle sur l'état de charge du système et lorsque la distance entre deux noeuds de la machine est importante dans l'actualisation de telle information.
Nous avons défini dans SIMAD plusieurs grilles de processeurs de tailles: 6x6, 8x8, 10x10,15x15 et 20x20. Ce type de topologie nous a également permis d'avoir une

bonne appréciation visuelle des effets produits sur la machine par les algorithmes de placement. Les processeurs sont homogènes et leurs caractéristiques sont les suivantes:

- Capacité de calcul:1000 unités de calcul par unité de temps (uc/ut)
- Quantité de mémoire: 120000 unités de mémoire (um)
- Débit des liens: 10 unités de mémoire par unité de temps (um/ut)

5.2 Etude du comportement de l'Algorithme Evolutif

L'étude du comportement de l'algorithme Evolutif a été faite à partir de l'observation et de la comparaison des graphes obtenus après l'exécution dans SIMAD de certains programmes de test. Dans cette section nous étudions la performance de l'algorithme Evolutif en faisant varier les paramètres qui sont à la base de son fonctionnement. Comme il a été dit dans le chapitre 4 les paramètres principaux qui interviennent dans le fonctionnement de l'algorithme Evolutif sont: la taille maximale des Solutions de Placement, le temps maximum de validité des SP, le nombre de processus par niveau de charge, et le type d'opérateur de fusion utilisé.

Nous commencerons d'abord par l'étude du comportement de l'algorithme Evolutif en définissant plusieurs tailles maximales des SP. L'idée est de pouvoir déterminer pour quelles tailles des SP on utilise le plus de processeurs de la machine et on réduit les temps d'exécution.

Nous définissons ensuite plusieurs temps maximaux de validité des SP et étudions leur influence sur les performances. Ce test est une aide pour déterminer quel est le temps maximal de validité des SP, adéquat pour exécuter un certain programme sous l'angle du nombre de créations effectué par les processus.

Pour continuer cette étude nous allons observer comment le nombre de processus par niveau de charge influence le nombre de ressources utilisées et l'équilibrage de la charge sur la machine. Nous avons utilisé deux types de niveaux de charge: dans le premier cas le nombre de processus par niveau de charge est fixé mais dans le deuxième cas ce nombre peut changer au cours de l'exécution.

Le dernier paramètre à étudier sera l'opérateur de fusion qui, dans le sens propre du terme, n'est pas vraiment un paramètre avec une valeur numérique associée, mais sa

définition influence les performances de l'algorithme Evolutif. Nous présenterons une comparaison entre deux types de fusion: La fusion qui forme des SP avec des noeuds qui ont le même niveau de chargé et la fusion qui forme des SP avec des noeuds qui ont différents niveau de charge et qui sont ordonnés de manière croissante par rapport à leur niveau de charge.

5.2.1 La taille des Solutions de Placement

La taille d'une SP est importante car elle est en relation avec la fréquence d'actualisation de la connaissance possédée par chaque noeud sur l'état de charge de la machine. Comme il a été expliqué dans la section 4.3.2, un noeud X, qui a un niveau de charge plus grand que le niveau de charge de sa SP et qui doit placer un nouveau processus, choisit un noeud Y dans sa SP pour lui faire une requête de placement. Si le noeud destinataire Y refuse la requête alors le noeud X enlèvera le noeud Y de sa SP (application de l'opérateur de déclin). Si après plusieurs refus la SP du noeud X devient vide, l'opérateur de fusion doit être appliqué pour reconstruire une nouvelle SP du noeud X.

Le choix de donner une taille N très grande aux SP implique N applications de l'opérateur de déclin pour que la SP devienne vide. Cela a une influence sur la validité de l'information possédée car dans ce cas là l'actualisation des SP n'est pas très fréquente. C'est après l'enlèvement de tous les noeuds que la SP est reconstruite par l'application de l'opérateur de fusion.

Par contre si nous choisissons des tailles très petites pour les SP, l'information possédée par chaque noeud sur l'état de charge peut être plus représentative de l'état actuel mais les SP deviennent vides plus rapidement. La reconstruction fréquente des SP peut provoquer un sur-coût dû aux envois des messages nécessaires pour cette reconstruction, pénalisant le temps total d'exécution de l'application.

Nous avons vu que la taille initiale des SP de chaque noeud est égale au nombre de voisins directement connectés (minimum 2 et maximum 4 pour une grille). Les SP peuvent grandir jusqu'à avoir une taille maximale T_{max}. La plus grande taille qu'une SP peut avoir est égale au nombre de noeuds dans la machine - 1, même si T_{max} a une valeur plus grande.

Si nous donnons un $T_{max} \geq$ *nb.total de noeuds* $-$ 1, nous laissons évoluer librement les tailles des SP, avec le risque d'avoir des SP très grandes pour certains noeuds. D'autre part, si nous voulons être surs que les tailles des SP ne seront pas

très grandes, nous pourrons donner à T_{max} une valeur plus petite que le nombre total de noeuds - 1.

Pour étudier l'influence des tailles maximales des SP sur les performances de l'algorithme Evolutif nous avons réalisé de nombreux test. Les programmes que nous avons utilisés appartiennent principalement au Groupe 1. Ces programmes ne génèrent pas beaucoup de messages et n'influencent donc pas de façon importante la charge sur les liens de communication provoquée par l'échange de messages entre processus. Au contraire, ils nous permettent d'étudier l'actualisation de l'information d'un noeud favorisant les transferts des SP sur les liens.

Le rapport entre la taille maximale des SP et le temps d'exécution obtenu ne peut pas être calculé avec précision. Dans les nombreux test réalisés nous avons fait exécuter plusieurs programmes sur différentes machines de tailles 6x6,8x8 et 10x10, en donnant aux SP les tailles maximales : 4, 6, 8, 10, 15, 20, 25, 30, 35, 40, 45 et 50.

Un des jeux de tests réalisés est constitué de 9 programmes du type "pparallel.prg" (programme numéro 4 du Groupe1). Ces programmes ont aussi un graphe de créations en forme d'arbre complet mais les processeurs pères font une quantité de calculs équivalent au temps d'exécution maximum de leurs fils.

Les variables des programmes ont été initialisées comme suit: quant_comp_feuille = 100000, capacite_calc_machine = 1000, temps_transfert_tache = 100, temps_transfert_donnees = 1000, temps_creation = 6 et le message qui annonce la terminaison d'un processus a un volume de 10000um. Les nombres de créations effectués par chaque processus varient de 2 à 10 et les hauteurs des arbres sont: 5 pour 2, 3, 4 et 5 créations, et 4 pour 6, 7, 8, 9 et 10 créations.

Nous montrons sur les figures 5.1 et 5.2 la variation des temps obtenus exécutant les 6 programmes qui font 2 (31 processus), 4 (341 processus), 5 (781 processus), 6 (259 processus), 8 (585 processus) et 10 (1111 processus) créations par processus. A côté de chacun des graphes de temps nous présentons le graphe associé qui indique le nombre maximum de noeuds utilisés à un moment donné de l'exécution.

Comme nous pouvons voir, dans ce test les temps d'exécution deviennent stables à partir d'une certaine taille des SP. Seulement au cas où le nombre de processus créés

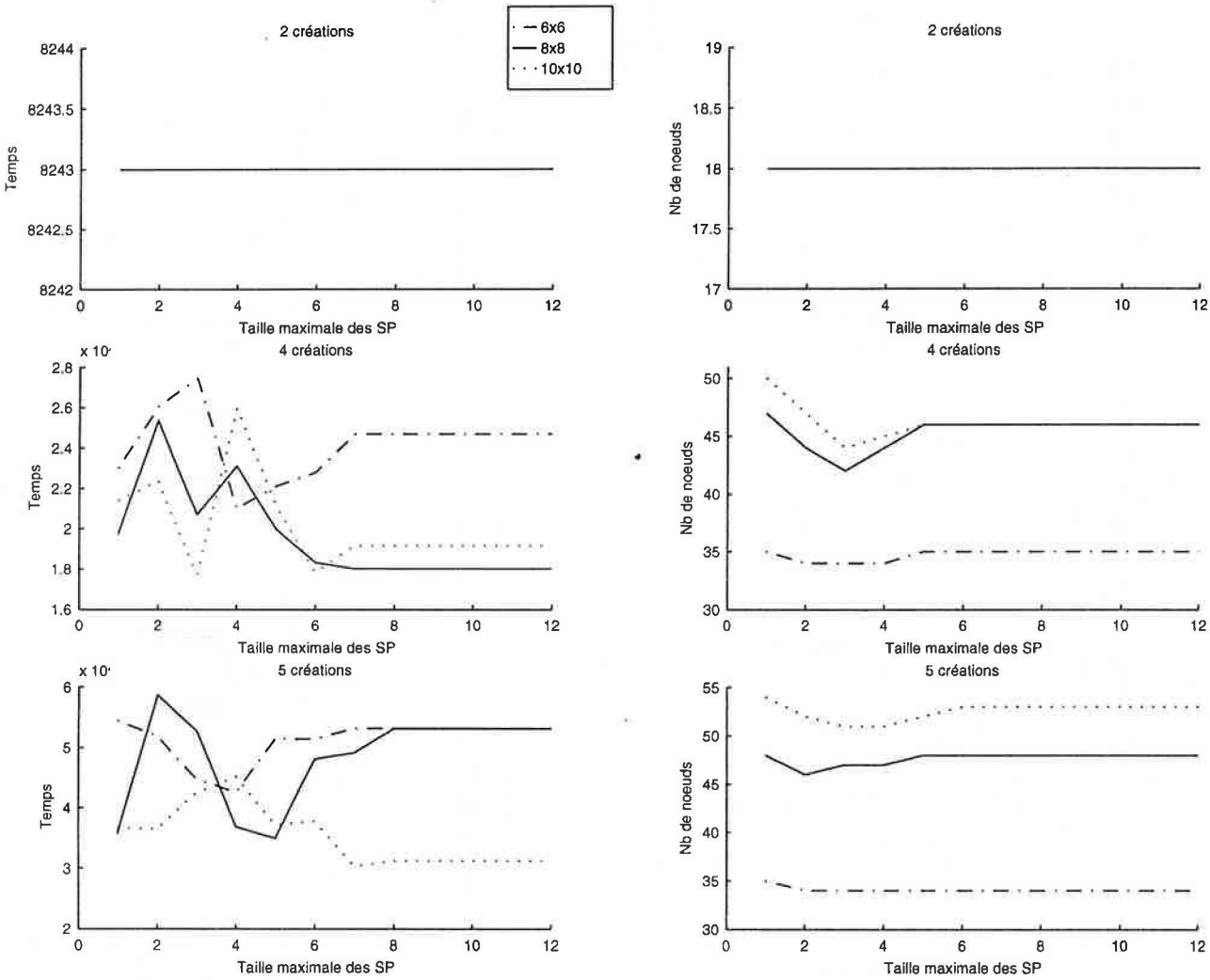


FIG. 5.1 - *Temps d'exécution et nombre de noeuds utilisés en fonction des tailles des SP (2, 4 et 5 créations par processus)*

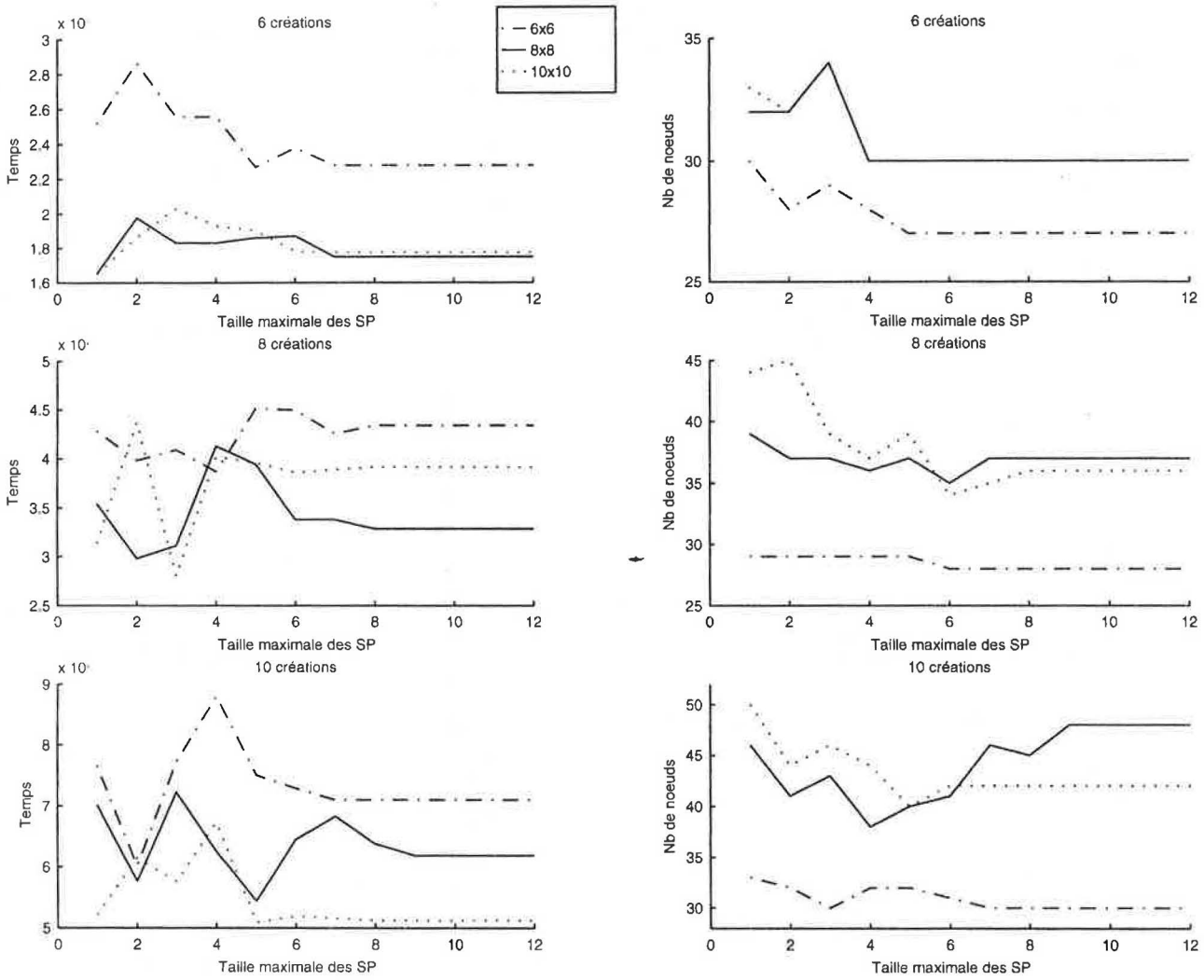


FIG. 5.2 - *Temps d'exécution et nombre de noeuds utilisés en fonction des tailles des SP (6, 8 et 10 créations par processus)*

dynamiquement n'est pas important, les SP évoluent peu et le temps d'exécution devient stable à partir des tailles plus petites que le nombre total de noeuds dans la machine.

Par exemple pour le programme où les processus pères créent seulement 2 autres processus et la hauteur de l'arbre est 5 (31 processus) les temps d'exécution deviennent stables à partir de la taille 4 des SP. Dans ce cas les créations effectués par les noeuds n'ont pas fait évoluer considérablement les SP et dans toutes les machines nous avons obtenu les mêmes temps d'exécution et le même nombre de noeuds utilisés.

Après avoir observé les résultats des tests réalisés, nous avons constaté que les temps arrivent à se stabiliser à partir d'une certaine taille maximale des SP. Le temps d'exécution dans la période stable n'est pas forcement le plus petit. Dans plusieurs cas, lorsque le programme est assez grand pour utiliser tous les noeuds de la machine, certaines tailles petites pourraient améliorer les temps d'exécutions obtenus dans le cas stable. D'autre part, si le programme est exécuté par une grille très grande où beaucoup de noeuds ne sont pas utilisés, les temps stables sont plus petits que ceux obtenus par l'utilisation de tailles plus petites des SP. Cette règle n'est pas générale mais peut être une aide dans le choix de la valeur des tailles maximales des SP.

5.2.2 Le temps de validité d'une Solution de Placement

Les SP sont valables pendant un certain temps. Comme nous avons expliqué dans la section 4.3.1, l'évolution des SP est déclenchée principalement par l'application des opérateurs de déclin et de croissance. Si un nœud n'a pas besoin d'appliquer ces opérateurs il gardera toujours la même SP et il risque de faire tomber le système dans un état de blocage s'il envoie toujours la même information.

Pour illustrer ceci, prenons le diagramme montré sur la figure 5.3 qui représente une partie d'une grille dont les noeuds ont initialement une charge nulle et la SP de chaque noeud contient les voisins directement connectés. Supposons que le noeud avec l'étiquette 0 est le seul à créer des processus de telle sorte qu'il fait d'abord augmenter son niveau de charge, et ensuite il cherchera à placer les processus suivants sur les noeuds de sa SP (les noeuds avec l'étiquette 1). Lorsque les noeuds 1 augmentent de niveau de charge ils n'accepteront plus de processus, et la SP du noeud 0 deviendra vide. C'est alors que le noeud 0 enverra des requêtes de SP aux noeuds 1 pour reconstruire une nouvelle SP. Grâce aux SP des noeuds 1, le noeud 0 pourra envoyer ses processus aux noeuds sous l'étiquette 2. De la même manière que

les noeuds 1, les noeuds 2 augmenteront leur niveau de charge et les SP des noeuds 1 et du noeud 0 deviendront vides. Après la reconstruction des SP des noeuds 1 et ensuite celle du noeud 0, les processus pourront être envoyés aux noeuds 3 jusqu'à ce qu'ils augmentent leur niveau de charge. C'est ici qu'on arrive dans un état de blocage car les noeuds 1 ne changeront plus de SP (ils ne font pas de créations qui font évoluer les SP) et ils garderont toujours les noeuds 3 dans leur SP, même si les noeuds 2 ont déjà dans leurs SP les noeuds 4 de la grille.

C'est pour cela qu'un temps maximal de validité des SP a été fixé. Cette solution résout le problème du blocage mais à la fois oblige à bien choisir le paramètre qui fixe ce temps maximal de validité d'une SP.

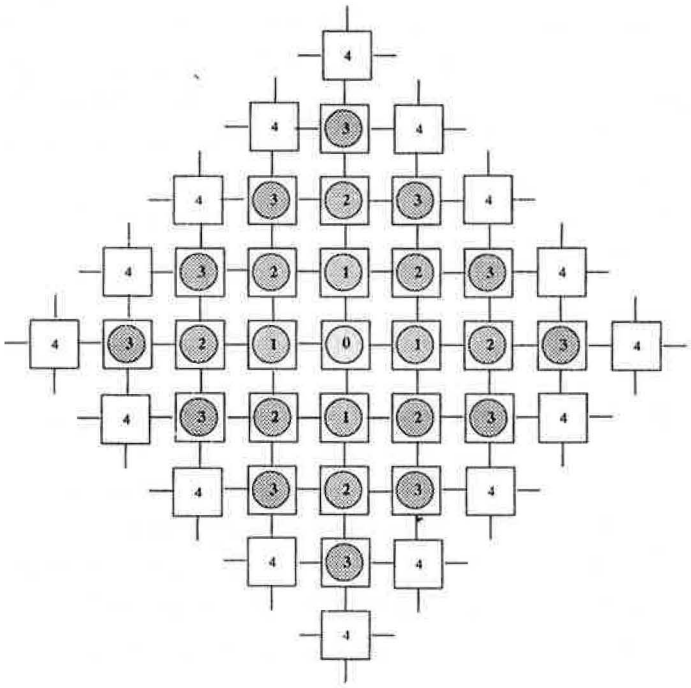


FIG. 5.3 - *Sites de placement pour le cas du blocage*

Si le temps de validité est très grand, le temps d'exécution de l'application peut être affecté considérablement car le temps mis pour l'actualisation des SP augmente et la qualité du placement se détériore. De l'autre côté, si le temps de validité est trop petit, le temps total d'exécution est aussi affecté car les liens de communications sont surchargés des messages nécessaires pour l'actualisation des SP.

Nous pouvons préciser que la validité d'une SP est toujours vérifiée lorsqu'un noeud reçoit une requête de sa SP. C'est seulement dans ce cas que la ré-actualisation d'une SP peut être faite par l'envoi des requêtes de SP aux voisins directement connectés. Si un noeud ne reçoit pas de requêtes de SP, il n'applique ni l'opérateur de déclin ni celui de croissance, il pourra conserver sa même SP pour longtemps comme dans l'exemple de la figure 5.3.

Pour étudier les effets de ce paramètre, nous avons utilisé un programme du groupe 2-2 dans lequel un seul processus initial p_i sera placé sur le noeud central de la machine et il sera le seul à créer N processus. Une fois que p_i a fait les créations il reste bloqué dans l'attente de N messages de synchronisation provenant de ses processus fils (messages qui servent à annoncer le placement réel des processus sur les noeuds).
Le processus p_i envoie ensuite un message à chaque processus pour qu'ils commencent leur travail qui consiste à communiquer avec d'autres processus comme indiqué sur la figure 5.4, pour N = 8 processus. Ce programme est nommé dans nos test: "fourier-parallel.prg", et son code général se trouve dans l'annexe B.6.

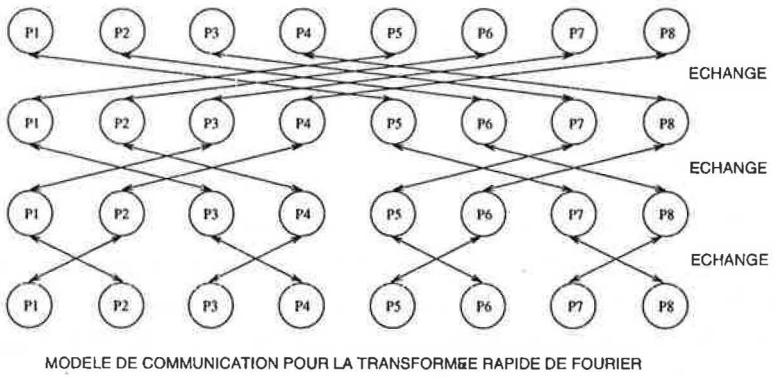


FIG. 5.4 - *Communications des processus du programme "fourier_parallel.prg"*

Après avoir fait exécuter ce programme sous les validités des SP: 5000, 3000, 1000, 600, 300, 200, 100, 80, 40 et 20 unités de temps, nous avons obtenu le graphe montré sur la figure 5.5. Ces résultats ont été obtenus en exécutant le programme "fourier-parallel" avec $N = 2^9 = 256$ processus. L'algorithme Evolutif a été initialisé avec 1 processus par niveau de charge, et une taille maximale des SP = 8 sur une grille de taille 8x8.

Premièrement on peut voir que le temps d'exécution du programme décroît lorsque le temps de validité des SP décroît aussi, cela continue jusqu'à arriver à une validité de 100 ut. A partir de 80 ut le temps d'exécution devient inversement proportionnel au temps des validités des SP.

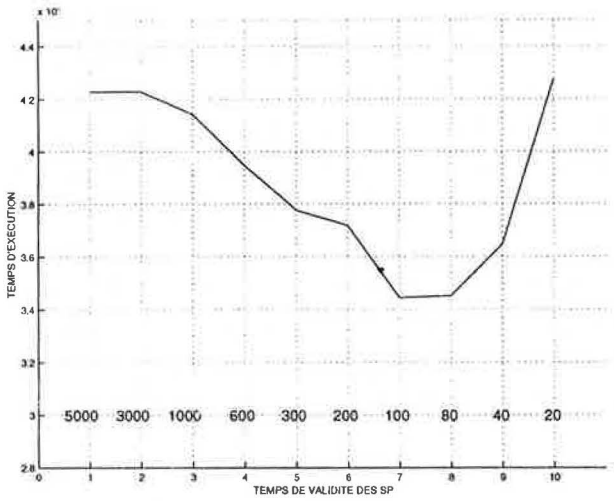


FIG. 5.5 - *Temps d'exécution en fontion de la validité des SP*

A partir de ces tests nous avons choisi certains temps de validité: 5000, 600, 100 et 40 pour visualiser comment les processus ont été placés. Sur la figure 5.10 nous avons les 4 distributions des processus associées à ces temps de validité. Le diagramme de la figure 5.6 montre une distribution des processus qui utilise seulement une petit partie des noeuds de la machine; pour ce test le temps de validité (5000 ut) n'a pas permis une rapide actualisation des SP.
Au fur et à mesure que le temps diminue, on trouve que le nombre de processeurs utilisés augmente comme on le voit sur les figures 5.7 et 5.8. Dans le cas où le temps de validité devient trop petit la distribution ne s'améliore pas; au contraire, la quantité de communications engendrée par l'actualisation des SP ne permet pas l'arrivée rapide des SP au noeud central qui veut placer des processus et il provoque une petit surcharge sur les noeuds du centre, comme indiqué sur la figure 5.9.

Pour des programmes de ce type où un seul processus crée tous les autres, le risque de blocage existe et le choix du temps de validité des SP est importante. Cependant

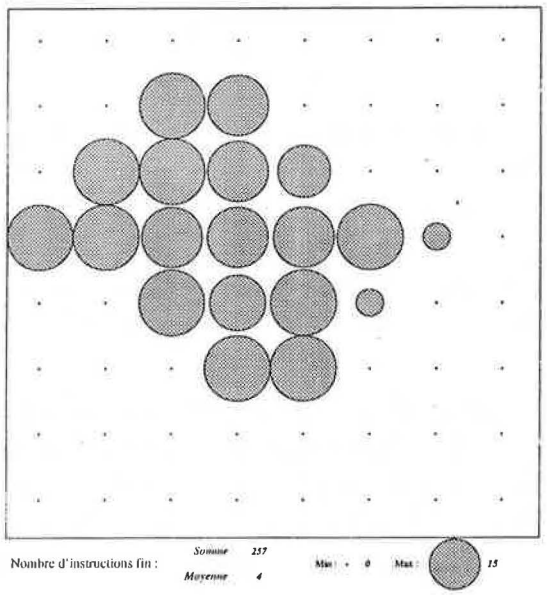


FIG. 5.6 - *Validité des SP = 5000 ut*

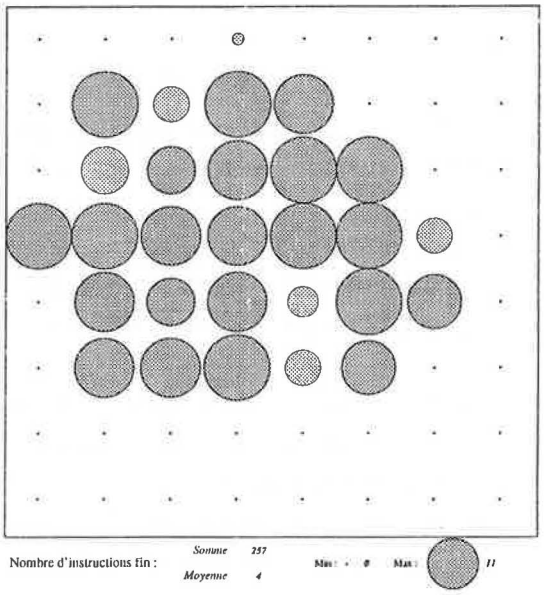


FIG. 5.7 - *Validité des SP = 600 ut*

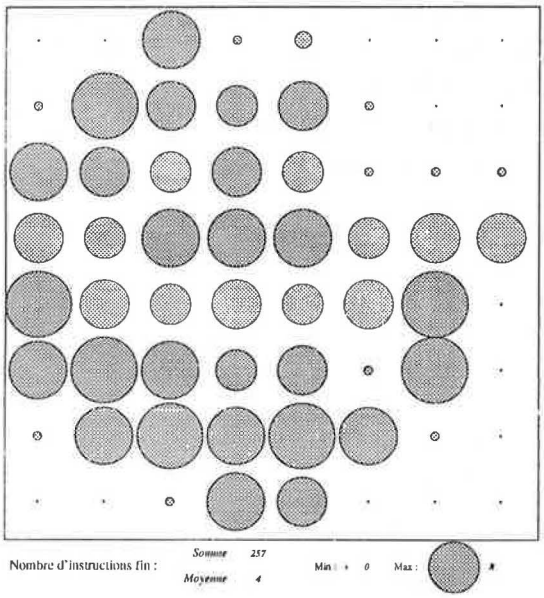


FIG. 5.8 - *Validité des SP = 100 ut*

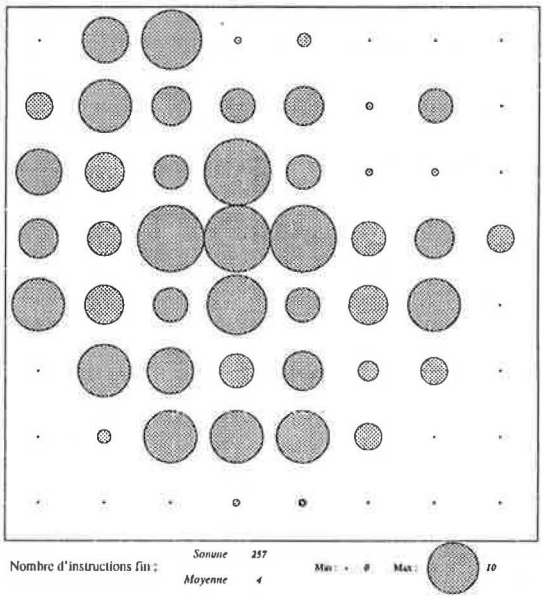


FIG. 5.9 - *Validité des SP = 40 ut*

FIG. 5.10 - *Nombre de processus exécuté sur chaque noeud sous différentes validités des SP*

pour des programmes où le graphe de créations est par exemple un arbre complet, l'évolution des SP ne se bloque pas car l'opérateur de fusion est appliqué constamment dans plusieurs noeuds de la machine. Nous avons sur la figure 5.11 les temps d'exécution d'un programme du groupe 2-1 avec un graphe de créations en forme d'arbre complet "arbre-complet-2.prg" (hauteur = 6, Nb de créations par processus = 4) par rapport au temps de validité des SP, exécuté sur une grille de taille 10x10.

Dans ce programme chaque fils fait un échange de messages avec son père (envoi et réception de données) avec une réalisation de calculs entre ces deux instructions. Finalement chaque fils envoie un dernier message à son père pour terminer son exécution (le programme général est dans l'annexe B.1 avec les variables nb_comm = 1, quant_comp = 100000, vol_comm = 1000, son_nomber = 4 et tree_height = 6).

Sur la figure 5.11, nous remarquons que les temps de validité des SP plus grands que 5000 (20000, 15000 et 10000) n'interfèrent pas dans le résultat total, ils laissent tout simplement la tâche de faire évoluer les SP aux opérateurs déclenchés par l'exécution du programme. Lorsque les temps deviennent plus petits il commencent influencer l'exécution du programme. Les temps 5000, 1000 et 500 ont fait augmenter le temps de simulation et à partir de 300 et 100 on trouve les temps le plus petits d'exécution, finalement il y a à nouveau une légère augmentation du temps que nous interprétons comme étant l'effet provoqué par la fréquente actualisation des SP.

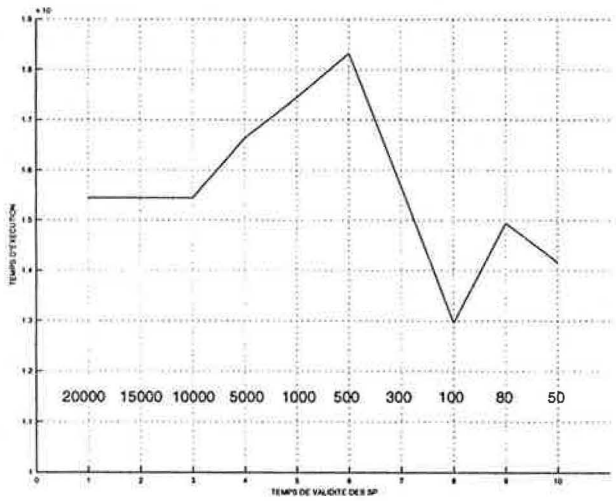


FIG. 5.11 - *Variation du temps d'exécution d'un programme avec un graphe de création en forme d'arbre complet, selon différents temps de validité des SP*

Après avoir réalisé plusieurs tests nous pouvons dire que le temps de validité des SP doit être fixé selon les caractéristiques des programmes. Si seulement quelques processus créent un grand nombre d'autres processus alors le temps de validité doit être petit. Par contre si les créations sont distribuées entre plusieurs processus (graphe de création en forme d'arbre), le temps de validité peut être grand pour ne pas surcharger les liens de communication.

5.2.3 Le nombre de processus par niveau de charge

Le nombre de processus par niveau de charge est sans doute un paramètre très important car il intervient dans la vitesse avec laquelle une Solution de Placement devient vide. L'idée de choisir un nombre petit de processus par niveau de charge a comme objectif de faire évoluer plus souvent les SP et de plus utiliser les ressources de la machine. Cependant, le fait de plus profiter des ressources n'implique pas forcement l'obtention des meilleures performances, car d'autres surcoûts (surtout de communication) pénalisent le temps d'exécution.

Pour étudier ce paramètre nous avons utilisé le programme "arbre-complet-2.prg" du groupe 2-1 avec un graphe de créations en forme d'arbre complet comme dans les tests antérieurs, la différence maintenant est que chaque père fait 2 échanges de messages avec ses fils, et finalement le père se bloque dans l'attente des dernières messages de terminaison des processus fils. Entre chaque échange de messages, les processus font toujours une même quantité de calculs (10000 uc), le programme général est montré dans l'annexe B.1, avec les variables quant_comp = 10000, vol_comm = 1000, son_nomber = 4, tree_height = 6 et nb_comm = 2.

La figure 5.12 montre les temps d'exécution de ce programme sur une grille de taille 10x10 en faisant varier le nombre de processus par niveau de charge. Nous pouvons voir que les temps les plus rapides d'exécution ont été obtenus en définissant 2 et 60 processus. La différence est liée à l'équilibrage de la charge sur les processeurs de la machine; comme le montrent les diagrammes de la figure 5.17 qui contiennent les quatre distributions des processus obtenues à partir de la définition de 1, 2, 60 et 150 processus par NvCh.

La distribution montrée sur la figure 5.13 correspond à la définition de 1 processus par NvCh, les noeuds ont été presque tous utilisés mais la constante séparation de pères et de fils a augmenté le temps total d'exécution dû au coût des communica-

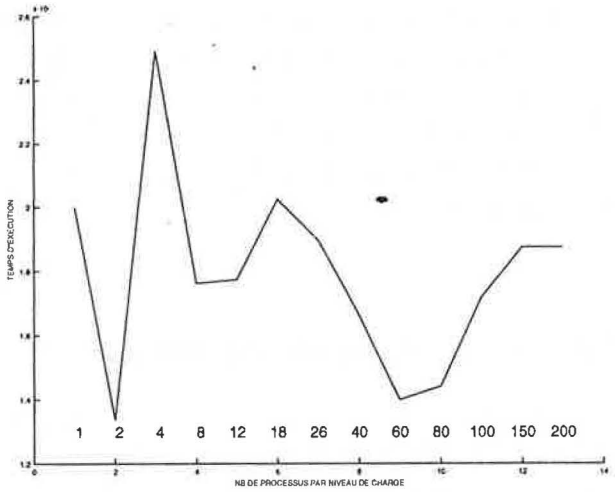


FIG. 5.12 - *Temps d'exécution en fontion du nombre de processus par NvCh*

tions rajouté.

Sur la figure 5.14 nous avons trouvé la meilleure distribution qui utilise 2 processus par niveau de charge et qui a obtenue le temps le plus petit d'exécution du programme "arbre-complet-2.prg". Dans le cas de 60 processus par NvCh (figure 5.15) on s'aperçoit que le prix de la réduction du temps d'exécution a été la sous-utilisation des noeuds de la machine. Finalement dans le cas extrême, on surcharge seulement quelques noeuds et le temps d'exécution augmente comme le montre la figure 5.16, après l'utilisation de 150 processus par NvCh.

L'idée d'utiliser un niveau de charge pour décider du placement des processus, peut être vue comme l'utilisation d'un système multi seuils. En général la plus grande partie des travaux utilisent 2 seuils de charge qui séparent les états de charge léger, moyen ou chargé. Dans l'algorithme Evolutif, un noeud peut augmenter son niveau de charge plusieurs fois jusqu'à devenir saturé. Si le nombre de processus par niveau de charge est très petit, un noeud peut atteindre un niveau de charge très élevé pour devenir saturé, mais au contraire, si nous définissons un nombre de processus très grand par niveau de charge, la saturation du noeud sera atteinte sous un niveau de charge petit.

Le fait de fixer un nombre de processus par niveau de charge donne de bons placements, suivant que l'on veut utiliser peu ou beaucoup des noeuds de la machine. Cependant pour rendre plus adaptable la distribution des processus nous avons im-

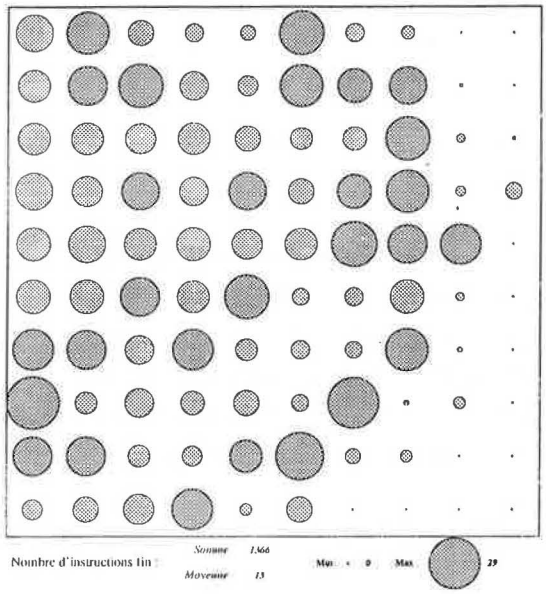


FIG. 5.13 - *1 processus par NvCh*

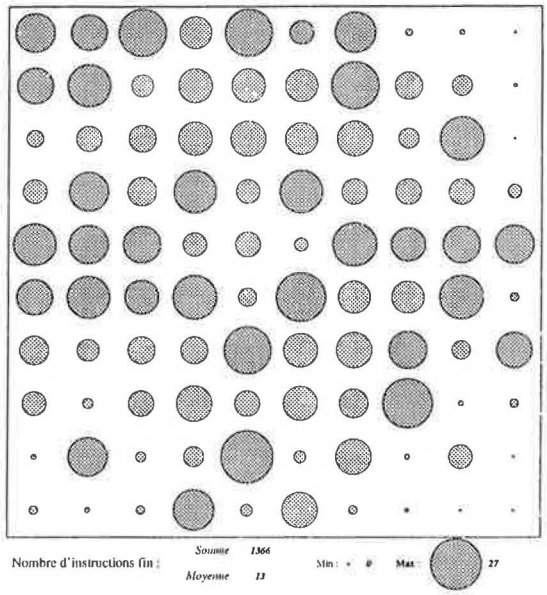


FIG. 5.14 - *2 processus par NvCh*

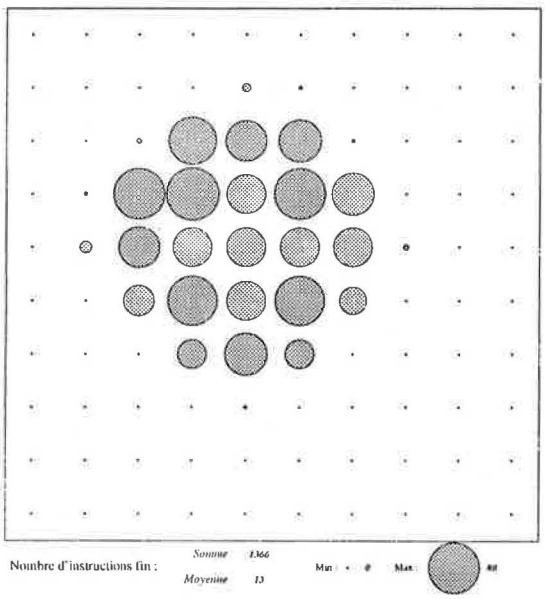


FIG. 5.15 - *60 processus par NvCh*

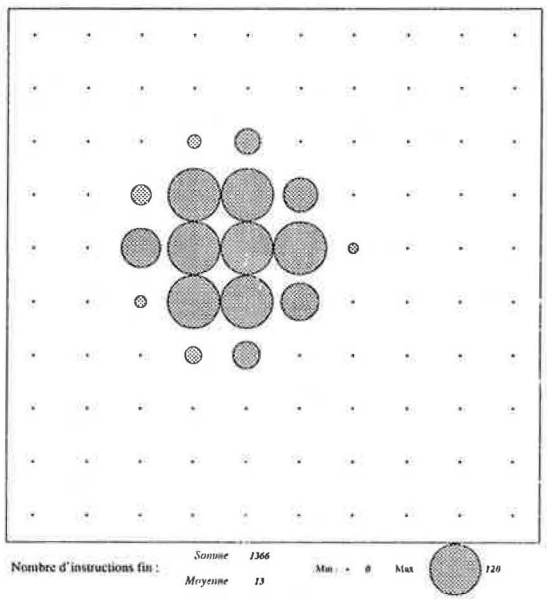


FIG. 5.16 - *150 processus par NvCh*

FIG. 5.17 - *Distribution des processus exécutés sur la machine avec le nombre de processus par niveau de charge= 1, 2, 60 et 150*

plémenté des niveaux de charge avec leurs nombres de processus qui changent au cours de l'exécution.

L'idée principale est d'assigner un nombre de processus élevé par niveau de charge aux noeuds qui n'ont pas beaucoup de processus, et au contraire, d'assigner un petit nombre de processus par niveau de charge aux noeuds qui en ont beaucoup, car le niveau de charge augmentera et ils n'accepteront plus d'autres processus. Chaque noeud de la machine recalculera le nombre de processus par niveau de charge lorsqu'il y a des créations ou bien des terminaisons de processus de la manière suivante:

Si (nombre de processus en mémoire >= Nb_max_de_processus_par_NvCh)
    Nb_de_processus_par_NvCh = 1;
sinon
    Nb_de_processus_par_NvCh =
        Nb_max_de_processus_par_NvCh - nombre de processus en mémoire;
NvCh = $\lfloor$*nombre de processus en mémoire / Nb_de_processus_par_NvCh*$\rfloor$;

Ainsi pour un nombre maximal de 8 processus par niveau de charge, l'évolution du nombre de processus par niveau de charge serait:

Nb de processus en mémoire	Nb de processus par NvCh	Niveau de Charge
1	7	0
2	6	0
3	5	0
4	4	1
5	3	1
6	2	3
7	1	7
>=8	1	>=8

Nous avons comparé les résultats donnés par le nouveau calcul adaptatif du nombre de processus par NvCh avec sa définition statique. En utilisant le même jeu de test que pour la version statique, nous avons observé que ce type de calcul a réduit le temps d'exécution en plusieurs occasions comme c'est montré sur la figure 5.18.

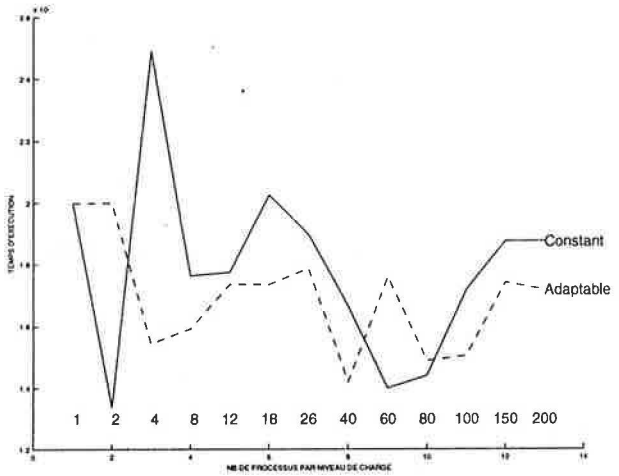


FIG. 5.18 - *Comparaison des niveaux de charge constant et adaptable exécutant le programme "arbre-complet-2.prg"*

Pour avoir d'autres comparaisons entre ces deux types de calcul (constant et adaptatif) du nombre de processus par niveau de charge, nous avons fait exécuter sur SIMAD un autre programme du groupe 1-1 nommé dans nos tests: "travaille-tjrs.prg" (voir annexe A.5, avec les variables feuille_comput = 180000, quant_comp = 10000, quant_comm = 10000, MEMOIRE = 1000, son_nomber = 2 et tree_height = 9) et en définissant pour l'algorithme Evolutif une taille maximale des SP = 6 et le temps de validité des SP = 50000 sur une grille de taille 10x10. Le graphe de créations du programme est aussi un arbre complet de hauteur 9 avec 2 créations par processus.

Le but du programme est que chaque processus, après avoir créé 2 processus, fera des calculs sans être bloqué à cause de l'attente des derniers messages de terminaison de ses fils. Pour cela il doit juste vérifier si les messages sont arrivés et si ce n'est pas le cas, il effectuera 10000 uc avant de répéter cette vérification. Si les processus fils sont placés loin, les processus pères vont peut être travailler un peu plus.

Nous pouvons observer sur la figure 5.19 les résultat de cette comparaison qui permet de voir que le calcul adaptable du nombre de processus par NvCh peut donner des temps d'exécution plus petits.

Pour avoir une dernière comparaison, nous avons fait exécuter sur une grille de taille 10x10 un programme du groupe 1-1 avec un haut degré de parallélisme. C'est à dire que plus les processus sont distribués sur différents noeuds, plus le temps d'exécution

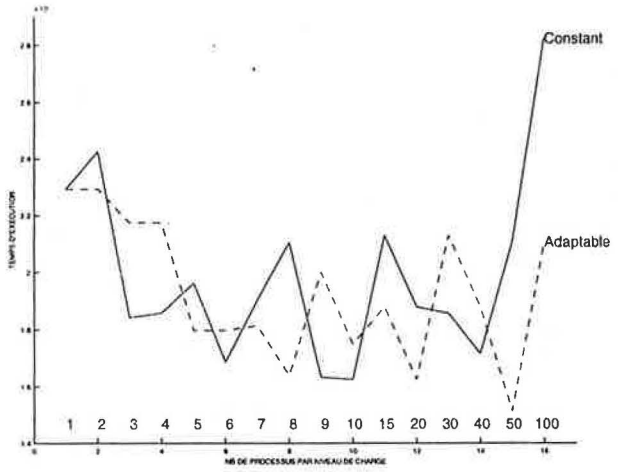


FIG. 5.19 - *Comparaison des niveaux de charge constant et adaptable exécutant le programme "travaille-tjrs.prg"*

est faible. Ce programme est nommé "pparallel.prg" et le code général se trouve dans l'annexe A.4 (avec les variables MEMOIRE = 1000, quant_comp_feuille = 1000000, son_nomber = 3, tree_height = 6, capacite_calc_machine = 1000, temps_transfert_tache = 100, temps_transfert_donnees = 1000, temps_creation = 6 avec un envoi de 10000 um). L'algorithme Evolutif a été initialisé avec une taille maximale des SP = 6 et le temps de validité des SP = 50000.

Pour ce test, La figure 5.20 montre les temps d'exécution obtenus en utilisant les deux types de calcul du nombre de processus par NvCh (Adaptable et Constant), les valeurs maximales définies pour le nombre de processus par NvCH varient entre 1 et 10. Nous pouvons voir que le calcul adaptatif donne des temps d'exécution égaux ou plus petits que l'utilisation d'un nombre de processus par NvCh constant. Finalement on extrait de ce jeu de test, la distribution des processus obtenue par la méthode constante et adaptative en définissant le nombre maximum de processus par NvCh = 4. Cette distribution est montrée sur la figure 5.21 sur laquelle on voit une meilleure utilisation des ressources pour la méthode adaptative.

Comme nous venons de le montrer, le nombre de processus par niveau de charge est un paramètre important qui peut être modifié au cours de l'exécution d'un programme par chaque noeud de la machine. Cette autonomie rend l'algorithme

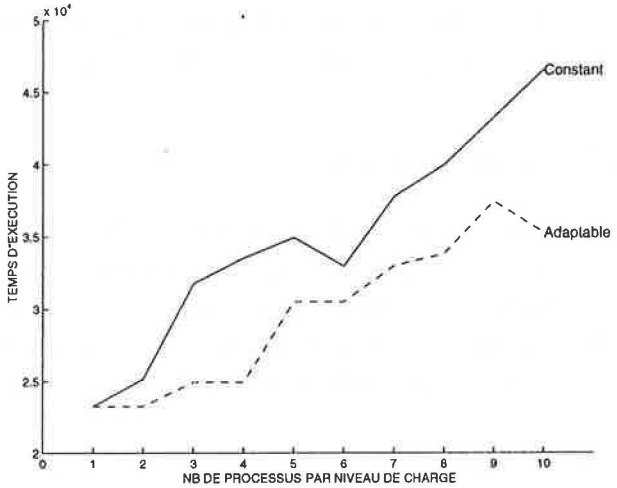


FIG. 5.20 - *Temps d'exécution du programme "pparallel.prg" en fontion du Nb de processus par NvCh*

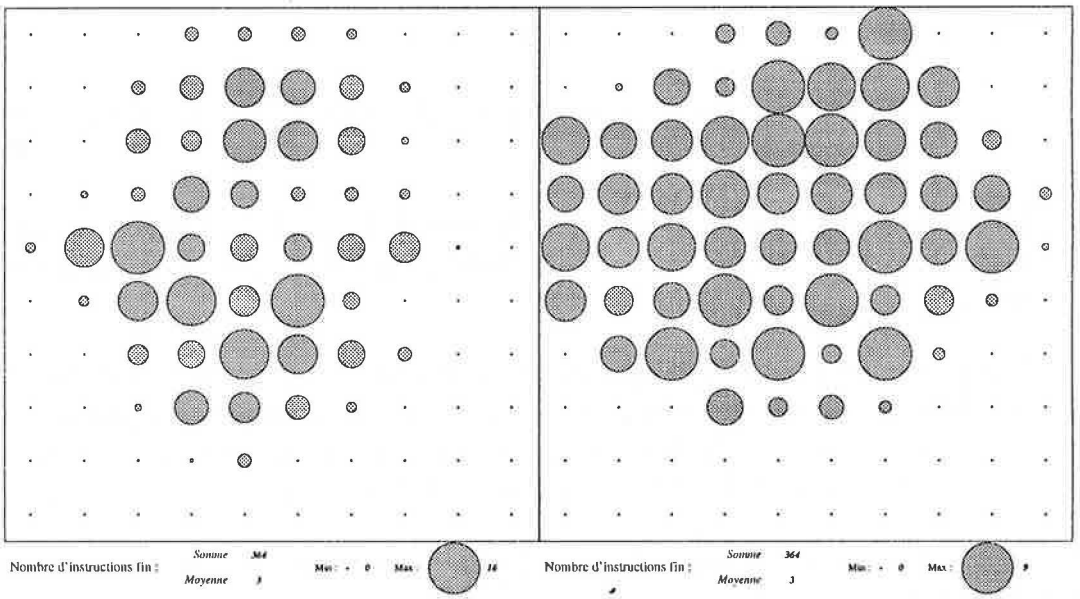


FIG. 5.21 - *Distribution des procesus obtenue par la méthode constante et adaptative (Nb Max Processus par NvCh=4)*

Evolutif adaptable vis à vis de l'adéquation du niveau de charge des noeuds aux changements de la charge sur la machine. D'une manière générale nous pouvons dire que si l'on cherche à utiliser le plus de noeuds possibles de la machine, nous devons utiliser un nombre de processus petit par niveau de charge.

5.2.4 L'opérateur de fusion

Le comportement de l'algorithme Evolutif dépend principalement des paramètres que nous venons de présenter. Cependant les opérateurs qui rendent possible l'évolution des SP peuvent aussi modifier les performances de l'algorithme. Dans cette partie nous faisons la comparaison des résultats obtenus en utilisant deux types de SP construites principalement par différents opérateurs de fusion.

Nous avions défini un type de fusion qui construit des SP homogènes (voir section 4.3.2), c'est à dire que les noeuds qui font partie d'une SP ont le même niveau de charge. Cependant il peut y avoir d'autres définitions de cet opérateur qui pourraient être plus simples. Dans cette section nous présentons une comparaison du type de fusion homogène utilisé dans les expériences antérieures (nommé dans cette section fusion1) avec la définition d'un autre opérateur de fusion (nommé fusion2) qui construit des SP dont les noeuds n'ont pas forcement le même niveau de charge mais les noeuds sont toujours ordonnés de façon ascendante par rapport au niveau de charge.

Pour mettre en place ce nouveau type de fusion les opérateurs de croissance et de déclin ont été définis de la manière suivante

- L'opérateur de croissance rajoutera le noeud Y dans la Solution de Placement SP_X du noeud X en conservant toujours l'ordre croissant des niveaux de charge.

- l'opérateur de déclin s'appliquera sur un noeud Y appartenant à la Solution de Placement SP_X du noeud X si le noeud Y n'accepte plus les processus du noeud X ou bien si SP_X dépasse sa taille maximale (effet de croissance) et le noeud Y a le niveau de charge le plus grand dans SP_X.

L'opérateur de fusion2 va simplement prendre les SP des voisins et il va en reconstruire une nouvelle avec les noeuds qui ont le niveau de charge le plus petit et ordonnés en ordre croissant. La taille maximale des SP est toujours respectée.

Pour comparer ces deux types de fusion, nous avons repris le dernier exemple présenté dans l'étude du calcul adaptatif et constant du nombre de processus par niveau

de charge en utilisant le programme "pparallel.prg" du groupe 1-1. La figure 5.22 illustre les temps d'exécution obtenus en utilisant la fusion1 avec le nombre de processus par NvCh adaptable (FEquAdapt) et constant (FEquConst) qui sont, en fait, les résultats obtenus précédemment sur le graphe de la figure 5.21. En plus sur la même figure nous avons les temps d'exécutions obtenus par la fusion2 avec le nombre de processus par NvCh adaptable (FOrdAdapt) et constant (FOrdConst). On peut observer que les temps les plus petits d'exécution sont donnés par l'utilisation du calcul adaptable du nombre de processus par NvCh, mais pour cet exemple la fusion2 n'a pas amélioré les résultats de la fusion1.

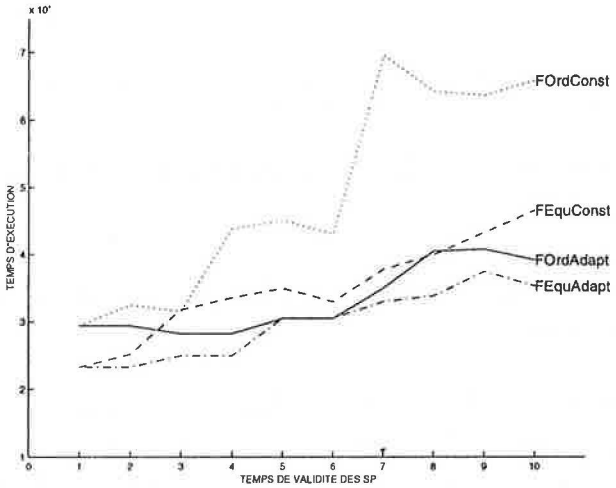


FIG. 5.22 - *Temps d'exécution du programme "pparallel.prg" en fonction de la fusion des SP*

Pour avoir une comparaison plus générale de ces deux types de fusions nous avons fait une dernière comparaison en exécutant un jeu de test qui utilise cinq types de programmes. Les programmes appartiennent aux groupes 1-1, 2-1 et 2-2 de programmes de test et sont les suivants:

Huit programmes du type "arbre-complet-1.prg" du groupe 1-1 (Annexe A.2), exécutés sur une grille de taille 8x8. Dans ce cas la capacité de mémoire des noeuds est 225000 um. Les huit programmes ont un graphe de créations en forme d'arbre complet de hauteur 5 avec 4 créations par processus. Les programmes diffèrent par :la

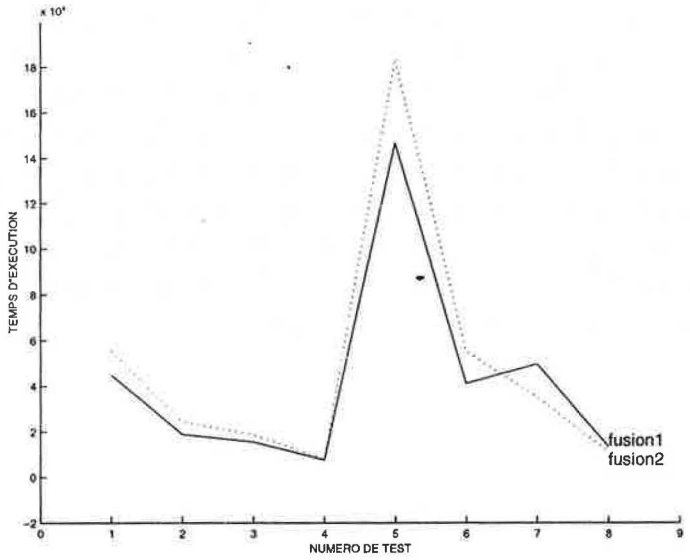


FIG. 5.23 - *Temps d'exécution de plusieurs programmes en fonction de la fusion des SP*

quantité de calculs effectués, le volume mémoire et le volume des communications, comme indiqué ci-dessous:

Num. de programme	*Volume Mémoire*	*Calculs*	*Volume Comm*	
1	1000	50000	1000	1 4 5
2	10000	50000	1000	1 4 5
3	1000	50000	10000	1 4 5
4	10000	50000	10000	1 4 5
5	1000	5000000	1000	1 4 5
6	10000	5000000	1000	1 4 5
7	1000	5000000	10000	1 4 5
8	10000	5000000	10000	1 4 5

Deux programmes du type "pparallel.prg" du groupe 1-1 (Annexe A.4), avec les variables:
MEMOIRE = 1000, quant_comp_feuille = 500000, capacite_calc_machine = 1000, temps_transfert_tache = 100, temps_transfert_donnees = 1000 et temps_creation = 6. La différence entre les deux programmes est le nombre de créations qui est 3 et 4 respectivement. La hauteur de l'arbre pour les deux cas est 6.

Deux programmes du type "travaille_toujours.prg" du groupe 1-1 (Annexe A.5), avec les variables:

MEMOIRE = 1000, feuille_comput = 200000, quant_comp= 10000, quant_comm = 10000. Dans les programmes le nombre de créations par processus est 3 et 4 respectivement avec une hauteur de l'arbre égale à 6.

Deux programmes du type "arbre-complet-2.prg" du groupe 2-1 (Annexe B.1), avec les variables:
MEMOIRE = 1000, quant_comp = 100000, vol_comm = 10000 et nb_comm = 2. Comme dans les programmes antérieurs ces deux programmes différent par le nombre de créations par processus: 3 et 4 respectivement. La hauteur de l'arbre est 6.

Deux programmes du type "espace-d-etats.prg" du groupe 2-2 (Annexe B.4), avec les variables: MEMOIRE = 1000, taille_tab = 6 pour le premier programme et 7 pour le second, vol_comm = 2500, nb_comm = 2 et calculs = 100000.

L'algorithme Evolutif a été initialisé de la manière suivante: 1 processus par niveau de charge (constant), taille maximale des SP = 6 et le temps de validité des SP = 50000 et en utilisant la grille de taille 10x10.

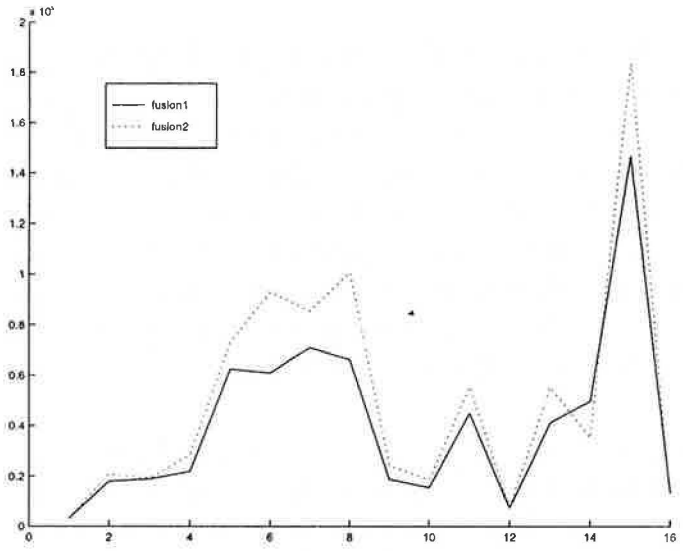


FIG. 5.24 - *Temps d'exécution de 16 programmes en fonction de la fusion des SP*

Nous pouvons voir sur la figure 5.24 les temps d'exécution des 16 programmes précédents obtenus en utilisant les deux types de fusion. D'une manière générale nous

trouvons que dans ces tests l'opérateur de fusion2 peut donner des temps d'exécution très proches à ceux de l'opérateur de fusion1 mais dans la plus grande partie des tests il n'y a pas d'amélioration du temps d'exécution obtenu. Nous avons donc retenu pour nos tests le type de fusion1 qui a donné les meilleurs résultats et qui ne demande pas au processeur de temps supplémentaire pour ordonner les noeuds dans les SP.

5.3 Etude comparative de l'algorithme Evolutif

Dans cette deuxième partie l'évaluation de l'algorithme Evolutif est faite par comparaison avec d'autres méthodes de placement dynamique de charge. Deux types de méthodes, avec des caractéristiques opposées, ont été utilisées dans la comparaison avec l'algorithme Evolutif.

5.3.1 Présentation des méthodes de placement dynamique de charge

La première méthode est celle du Gradient [LK87], implantée sur SIMAD. La méthode du Gradient est adaptée à la régulation de la charge se basant sur la notion de proximité du noeud le moins chargé. Le deuxième type de méthode est une méthode aveugle où chacun des noeuds où seront transférés des processus est choisi au hasard entre les N plus proches voisins. Si après un certain nombre de choix aucun des N noeuds n'accepte plus de processus, le prochain choix se fait entre tous les noeuds de la machine. Dans cet algorithme on prend en compte aucune information pour décider de transférer un processus. Trois algorithmes de ce type on été utilisés avec $N = 4, 25$ *et* M, où M = nb. total de processeurs.

En général, la comparaison de plusieurs algorithmes d'allocation dynamique n'est pas simple. Il est préférable que deux algorithmes soient exécutés sur la même plateforme d'évaluation pour obtenir une comparaison représentative. SIMAD nous a permis d'obtenir une telle comparaison des algorithmes de placement sous les conditions suivantes d'exécution: un processus est exécuté par un seul processeur, la quantification de la charge est la même pour tous les algorithmes et ils n'ont pas une connaissance préalable sur le comportement des applications.

Dans la suite, nous allons décrire les paramètres nécessaires associés à la définition de chacune des méthodes devant être comparées. Les valeurs de ces paramètres sont

déterminés avant chaque simulation, avec l'idée d'exécuter au mieux les deux groupes de programmes de test

- Aléatoire-4 et Aléatoire-25

  Cet Algorithme choisit un noeud au hasard pour placer un processus. Le choix se fait entre les M (M=4 ou 25) noeuds voisins les plus proches. Un noeud peut refuser la requête de placement d'un autre noeud seulement dans le cas où il n'a plus d'espace mémoire pour accueillir le processus. Pour cet algorithme il n'y a pas de paramètres précis qui règlent son comportement.

- Gradient

  La méthode du gradient a un fonctionnement à double seuil. Lorsqu'un noeud est légèrement chargé il peut recevoir des processus provenant d'autres noeuds. Si le seuil de charge devient moyen le noeud ne reçoit plus de processus externes mais il exécute les processus qu'il possède et les processus qui pourraient être générés par ceux-là. Si le seuil devient chargé le noeud ne pourra exécuter ni des processus externes ni des processus crées localement, alors il essayera de transférer sa charge vers d'autres noeuds. Les deux paramètres à régler sont:

  - Le seuil moyen
  - Le seuil chargé

  La mesure de la charge locale d'un noeud est faite en prenant en compte le nombre de processus, en état d'attente et en état d'exécution, sur le noeud. Le seuil moyen et le seuil chargé contiendront donc des valeurs entières. La valeur de ces seuils dépendra des caractéristiques générales des applications et des machines utilisées.

- Evolutif

  Lorsqu'on a étudié le comportement de l'algorithme Evolutif (section 5.2), nous avons présenté plusieurs paramètres qui règlent son comportement:

  - La taille maximale des Solutions de Placement
  - Le temps de validité d'une Solution de Placement
  - Le nombre de processus par niveau de charge
  - L'opérateur de fusion

Dans l'algorithme Evolutif nous considérons la même mesure de la charge que celle utilisée pour la méthode du Gradient (le nombre de processus, en état d'attente et en état d'exécution, sur le noeud). Pour réduire le nombre de comparaisons obtenues en faisant varier les paramètres des méthodes, nous avons utilisé seulement le type de fusion1 qui construit des SP contenant des noeuds avec le même niveau de charge.

Dans nos tests la performance d'un algorithme de placement sera associé à trois facteurs, dans l'ordre suivant: le temps d'exécution obtenu, le nombre de noeuds utilisés et l'équilibrage de la charge.

Cette étude comparative est divisé en trois parties. Les deux premières parties sont consacrées à la comparaison de l'algorithme Evolutif avec les méthodes du Gradient et l'Aléatoire pour l'exécution des deux groupes principaux de programmes de test présentés dans la section 3.10 et au début de ce chapitre. Pour chaque groupe de programmes nous présentons une comparaison générale et ensuite nous montrons quelques exemples particuliers qui font apprécier les performances de l'algorithme Evolutif.
La dernière partie donne une comparaison générale des trois algorithmes exécutant quelques uns des programmes de test avec différentes initialisations du simulateur.

5.3.2 Comparaison sur le groupe 1 de programmes qui font peu de communications

Dans cette partie nous présentons une comparaison des performances des trois algorithmes de placement Aléatoire-4, Gradient et Evolutif, exécutant les programmes de test du groupe 1. Nous avons défini deux plans d'expériences, le premier utilise les programmes avec un graphe de créations en forme d'arbre complet (programmes du groupe 1-1) et le deuxième utilise les programmes avec d'autres graphes de création (appartenant au groupe 1-2).

5.3.2.1 Exécution des programmes avec un graphe de créations en forme d'arbre complet

Le plan d'expériences associé à cette comparaison est formé par 72 programmes de test. Chacun des programmes correspond à un des 6 types de programme du groupe 1-1.

Les 12 premiers programmes sont du type du programme 1 "processus-indep.prg" où chacun des processus créés fait la même quantité de calculs et ensuite envoie un

message de terminaison à un processus initial. Les 12 programmes ont les caractéristiques suivantes:
Volume mémoire du processus initial = 1000um, volume mémoire des processus de l'arbre = 5000um, quantité de calculs à effectuer = 85000uc et volume du message envoyé au processus initial = 1000um.
Les programmes diffèrent par le nombre de processus créés:

Num. de programme	*Nb.Créations par processus*	*Hauteur de l'arbre*	*Nb.Total de processus*
1	2	10	1024
2	3	7	1094
3	4	6	1366
4	5	5	782
5	6	5	1556
6	7	4	401
7	8	4	586
8	9	4	821
9	10	4	1112
10	20	3	422
11	25	3	652
12	30	3	932

De la même manière nous avons utilisé 12 autres programmes du type du programme 2 "arbre-complet-1.prg" qui a presque le même comportement que le programme 1, sauf dans l'envoi du dernier message qui est dirigé au processus père au lieu du processus initial.
Les 12 programmes de ce type ont les caractéristiques suivantes: volume mémoire du processus initial = 1000um, volume mémoire pour les processus de l'arbre = 5000um. La quantité de calculs à effectuer par les processus = 125000uc et le dernier message envoyé au père a un volume = 1000um. Ces programmes (numérotés du 13 au 24) diffèrent par le nombre de processus créés comme dans le cas antérieur.

Pour chacun des 4 types des programmes restants du groupe 1-1, nous avons également utilisé 12 programmes qui diffèrent par le nombre de processus créés comme dans les programmes antérieurs. Dans la suite nous allons seulement décrire ses caractéristiques générales.

Les 12 programmes du type du programme 3 "prg-all-1.prg" (numérotés du 25 au 36) ont les caractéristiques suivantes: volume mémoire des processus = 5000um, quantité de calculs effectué par le processus racine = 1800000uc, la quantité de calculs effectués par les autres processus est celle faite par leurs pères moins 150000uc. Le dernier message est envoyé aux pères et a un volume = 1500um.

Pour les programmes numérotés du 37 au 48 (du type du programme 4 "pparallel.prg") les caractéristiques sont: volume mémoire des processus = 5000um et le message envoyé à la fin de l'exécution a un volume = 10000um. En prenant en compte les caractéristiques de la machine, pour calculer les quantités de calcul effectués par chaque processus nous avons initialisé les variables du programme comme suit: quant_comp_feuille = 100000uc, capacité_calc_machine = 1000uc/ut, temps_transfert_tache = 500ut et temps_transfert_donnees = 1000ut.

Les programmes 49 au 60 sont du genre du programme 5 "travaille-tjrs.prg" avec les caractéristiques suivantes: volume mémoire des processus = 5000um, la quantité de calculs effectués par les processus qui sont des feuilles dans l'arbre = 15000uc et le volume du message envoyé aus pères = 500um. Chaque processus racine répète 3000uc si les messages de terminaison de ses fils ne sont pas tous arrivés.

Finalement pour les programmes 61 au 72 du type du programme 6 "arbr1-AuDiAuDi.prg" les caractéristiques sont: volume mémoire des processus = 5000um, la quantité de calculs faite par le processus racine = 2500000uc, les autres processus font la même quantité de calculs faite par leurs pères moins 17000uc. Le dernier message envoyé aux pères a un volume = 10000um.

En ce qui concerne l'initialisation des paramètres des algorithmes de placement, pour les programme du type 1 et 2 ("processus-indep.prg" et "arbre-complet-1.prg") les deux seuils de la méthode du Gradient sont: 1 processus pour le seuil moyen et 3 pour le seuil chargé. C'est à dire que l'état de charge légère d'un noeud implique un nombre de processus ≤ 1, la charge moyenne permet un nombre de processus ≤ 3 et l'état chargé s'obtient lorsque le nombre de processus sur un noeud > 3. Pour les programmes des types 3 au 6 ("prg-all-1.prg", "pparallel.prg", "travaille-tjrs.prg" et "arbr1-AuDiAuDi.prg") les seuils de la méthode du Gradient ont eu les valeurs: 3 et 5 processus respectivement.

Les paramètres de l'algorithme Evolutif ainsi que ceux de la méthode du Gradient ont été choisis en cherchant à utiliser le plus de noeuds de la machine. A cet effet, les paramètres de l'algorithme Evolutif ont eu les valeurs suivantes: le nombre de processus par niveau de charge = 1, la taille maximale des SP = 6 et le temps maximum de validité d'une SP = 50000 ut.

Nous avons fait exécuter ces 72 programmes sur une grille de taille 10x10 et en utilisant les trois algorithmes de placement: Aléatoire-4, Gradient et Evolutif. Sur la figure 5.25 se trouvent les graphes qui représentent le nombre maximum de noeuds

utilisés à un moment donné, pour chaque algorithme de placement.

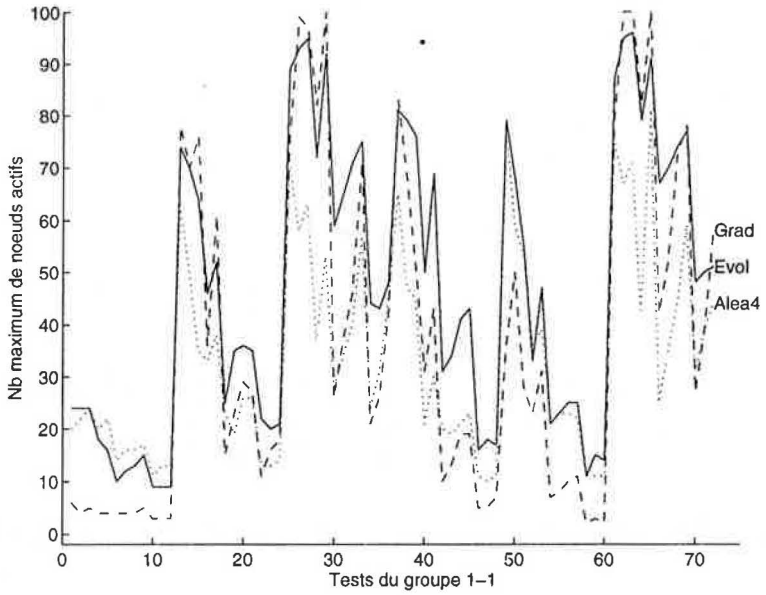


FIG. 5.25 - *Nombre maximum de noeuds actifs à un moment donné*

Dans ce plan d'expériences nous avons observé que l'algorithme Evolutif a obtenu, de manière générale, les plus grand nombres de noeuds utilisés à un moment donné. L'algorithme du Gradient dans certains tests (13-15, 25-30 et 61-65) arrive aussi à utiliser le plus de noeuds de la machine. En ce qui concerne l'algorithme Aléatoire-4, il a pu améliorer les autres méthodes seulement pour l'exécution de quelques programmes du type 1 (3-12) où les processus pères ne se bloquent pas en attendant les messages de terminaison de leurs fils.

Nous pouvons voir sur la figure 5.26 le nombre maximum de processus exécutés par un noeud pour les trois algorithmes de placement. Ce nombre est un indicateur qui peut estimer la qualité de l'équilibrage de la charge obtenu sur la machine.

A première vue on peut apercevoir que la méthode du Gradient a donné beaucoup de travail à quelques noeuds de la machine, surtout pour les test 1-12 et 48-61. D'autre part, on peut voir que l'algorithme Evolutif a pu améliorer, dans la plus

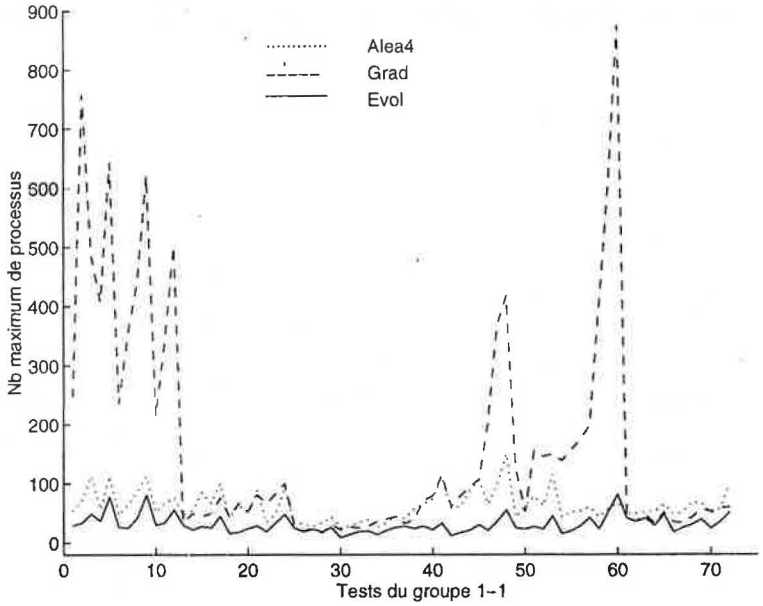


FIG. 5.26 - *Nombre maximum de processus exécutés par un noeud*

grande partie des tests, l'équilibrage de la charge par rapport à celle obtenue par les algorithmes Aléatoire4 et celui du Gradient, donnant le plus petit nombre maximum de processus exécutés par un noeud.

Finalement, le graphe de la figure 5.27 montre les temps d'exécution des programmes obtenus par les différents algorithmes de placement. Pour le premier type de programme où il n'y a pas d'attente de messages de la part des processus pères, les algorithmes Evolutif et Aléatoire ont donné les temps le plus petits d'exécution. A partir des tests 13 au 60 des programmes où les processus pères se bloquent en attendant les dernières messages de leurs fils, l'algorithme Evolutif a donné, d'une manière générale, les temps le plus petits d'exécution.
Pour les derniers tests où les programmes font exécuter 2 fois un arbre de processus sur la grille (mettant à œuvre la qualité de l'actualisation de l'information possédée par chaque noeud), les algorithmes Evolutif et du Gradient ont obtenu les temps le plus petits d'exécution.

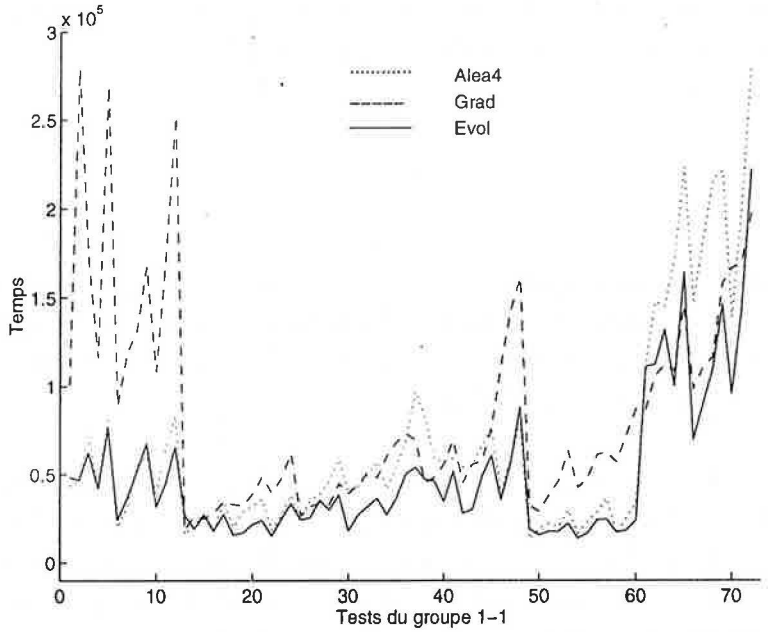


FIG. 5.27 - *Temps d'exécution pour les programmes du groupe 1*

5.3.2.2 Quelques exemples particuliers

Nous avons utilisé deux programmes du groupe 1-1 pour donner deux exemples de comparaison des trois méthodes de placement d'une manière plus détaillée. Dans la suite nous présenterons ces exemples qui utilisent des programmes du type "travaille-tjrs" et "pparallel.prg".

Le premier exemple fait une comparaison des algorithmes Aléatoire-4, du Gradient et l'Evolutif pour l'exécution d'un programme du type "travaille-tjrs" sur une grille de taille 8x8. Dans ce programme la hauteur de l'arbre est égale à 6 et chaque processus crée 3 autres processus (nb. total de processus = 364). Un processus père fait de calculs jusqu'à recevoir tous les messages de terminaison de ses fils.
La quantité de calculs effectuée par chaque processus père à chaque fois que les messages ne sont pas encore arrivés est de 20000uc et la quantité de calculs effectuée par les processus qui sont les feuilles de l'arbre est de 2800000 uc.

Cette expérience a pour objectif d'étudier le comportement des algorithmes d'équilibrage dynamique de charge vis à vis du choix des sites d'exécution d'un processus père et de ses fils. Si certains processus d'un programme passent beaucoup de temps à faire de calculs, cela peut s'interpréter comme s'ils ont beaucoup attendu l'arrivée des messages de terminaison de leurs fils. Si la méthode de placement utilisée éloigne

beaucoup un processus fils de son père, ou bien le met sur un noeud très chargé, les messages de terminaison vont tarder à arriver à leur destination et le temps total d'exécution augmentera.

Le diagramme supérieur gauche de la figure 5.28 montre la distribution des processus obtenue en utilisant la méthode aléatoire qui choisit au hasard un processeur d'entre les 4 voisins pour placer un processus. C'est cette méthode qui a eu le temps d'exécution le plus élevé (67462 ut).

Sur la même figure 5.28 dans le diagramme supérieur droit on peut voir la distribution des processus obtenue après l'utilisation de la méthode du Gradient. Les seuils ont été fixés de telle façon qu'on puisse utiliser le plus grand nombre de processeurs de la machine (Charge Légère: 4 processus, Charge Moyenne: 5 processus et Charge Lourde: 6 processus). Le temps d'exécution a été amélioré par cette deuxième méthode (34768 ut).

Finalement, pour ce test nous avons le diagramme de la partie basse de la figure 5.28 qui représente la distribution des processus obtenue après l'utilisation de l'algorithme Evolutif. Pour être en accord avec l'idée d'utiliser le plus grand nombre de processeurs, on a fixe les paramètres de cette méthode comme suit: 1 processus par niveau de charge, la taille maximale d'une SP est fixée à 6 processeurs avec un temps minimal d'actualisation des SP de 50000 ut. Le temps d'exécution simulé a été à nouveau amélioré par l'algorithme Evolutif (25671 ut).

Sur la figure 5.29 nous avons les graphiques représentant l'évolution dans le temps du nombre de processeurs utilisés et du nombre de messages sur la machine, obtenus par les trois méthodes. On peut observer que malgré le nombre élevé de messages engendrés par l'algorithme Evolutif (nécessaires pour la reconstruction des SP), le placement obtenu a donné le plus petit temps d'exécution.

Comme dernier résultat de cette comparaison, la figure 5.30 montre les quantités de calculs effectuées par chaque processeur selon la méthode de placement utilisée. Suivant les différentes tailles des boules et les différentes échelles, on peut voir que pour l'algorithme Evolutif les processeurs travaillent moins et de façon plus équitable que pour les méthodes Aléatoire et du Gradient. Les temps de calculs variant, entre

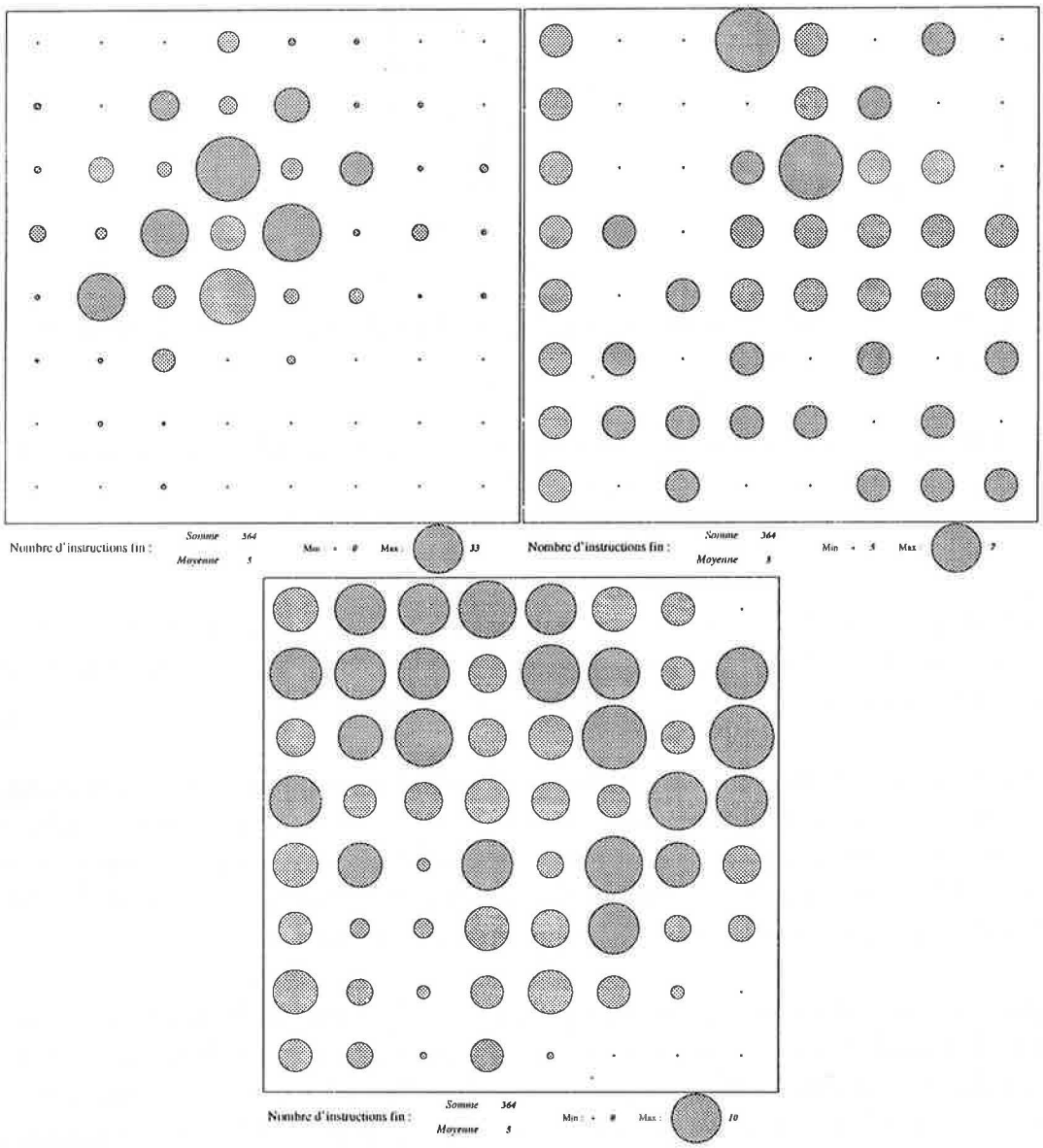


FIG. 5.28 - *Distribution des processus sur les processeurs. Aléatoire4 et Gradient (en haut) et Evolutif (en bas)*

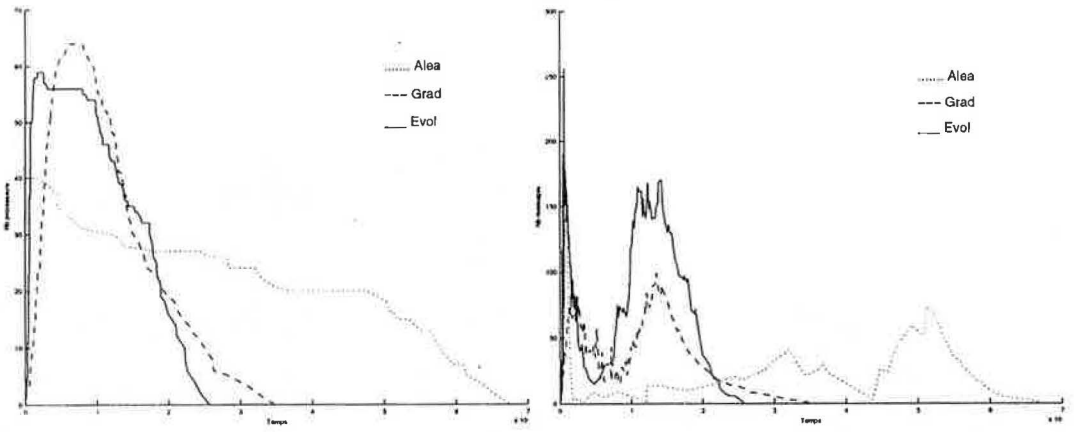


FIG. 5.29 - *Evolution du nombre de processeurs, du nombre de liens et du nombre de messages dans le temps*

0 et 63360 pour la méthode Aléatoire-4, entre 7400 et 30120 pour le Gradient et entre 0 et 23360 pour l'algorithme Evolutif.

Le deuxième exemple détaille la comparaison des trois algorithmes pour l'exécution d'un programme du type "pparallel.prg" où le temps de calcul des processus correspond au temps maximum d'exécution de leurs processus fils.

Si chaque processus était placé sur un noeud différent d'une machine complètement connectée (avec le nombre de processeurs égal au nombre de processus), lorsqu'un processus père aurait terminé ses calculs, il ne se bloquerait pas en attendant l'arrivée du dernier message de ses fils. Le temps total d'exécution serait donc le temps d'exécution du processus racine du programme (le cas idéal).

Le programme a été exécuté sur trois grilles de tailles 15x15,10x10 et 8x8. Ainsi le temps de transfert d'un processus sera calculé par rapport au volume du processus à transférer et le débit des liens de la machine (10um/ut). Les variables du programme sont initialisées comme suit: quant_comp_feuille = 100000uc, le graphe de créations de ce programme est un arbre complet de hauteur 7 et chaque processus crée 3 autres processus. D'autres variables sont: capacite_calc_machine = 1000uc/ut, temps_transfert_tache = 100ut, temps_transfert_donnees = 1000ut, volume mémoire des processus = 1000um et le volume des messages envoyés à la fin de l'exécution = 10000um.

Les temps d'exécution obtenus par les différentes méthodes de placement varient

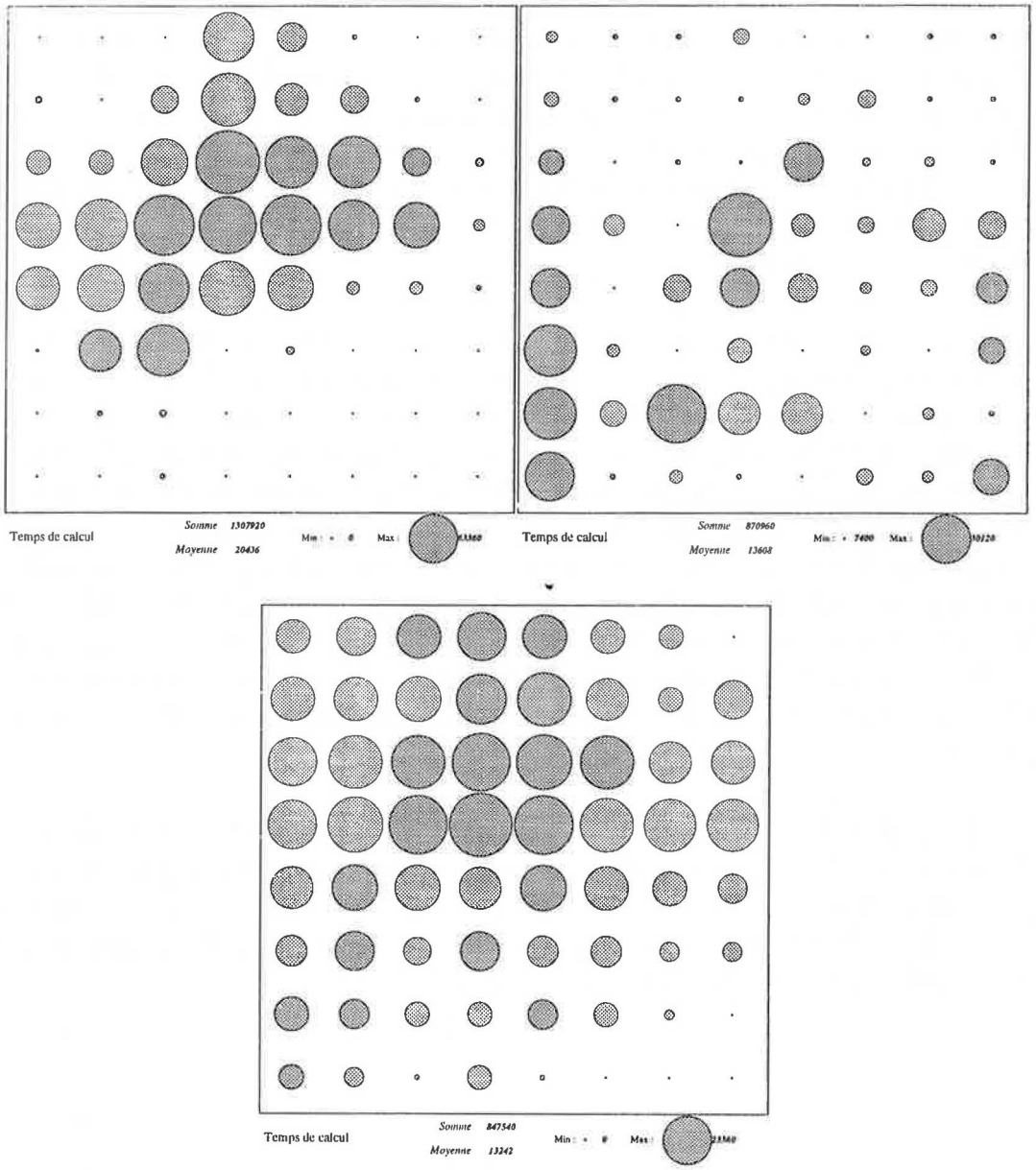


FIG. 5.30 - *Quantités de calcul effectuées par chaque processeur, Aléatoire-4, Gradient (en haut) et Evolutif (en bas)*

selon la machine sur laquelle elles s'exécutent. On peut voir sur la table 5.1 l'initialisation des paramètres des méthodes ainsi que leurs temps d'exécution et la variation du nombre de processus exécuté sur chaque noeud.

Dans cette comparaison l'algorithme Aléatoire-4 a donné les temps d'exécution les plus grands pour les trois machines. Pour ce type de programme il n'est donc pas convenable d'utiliser une méthode aveugle pour placer les processus. Nous avons observé que la Méthode du Gradient est meilleure que l'algorithme Evolutif lorsqu'on utilise des machines plus grandes (m15x15). Sur des machines de taille plus petite (m10x10 et m8x8), les résultats ont montré que l'algorithme Evolutif est plus performant que les autres.

L'explication de ce comportement est que les algorithmes du Gradient et l'Evolutif ont un comportement différent lorsque les processus sont placés vers les coins de la grille. Dans la méthode du Gradient, si un noeud chargé qui se trouve au coin de la grille veut placer un processus, et si les noeuds qui sont autour de lui sont aussi chargés, le processus sera placé plus loin (où il y a un noeud légèrement chargé). Cette contrainte est une conséquence de l'utilisation de 2 seuils. D'autre part, dans l'algorithme Evolutif, si un noeud chargé qui se trouve au coin de la grille veut placer un processus, et si les noeuds qui sont autour de lui ont le même niveau de charge, le choix du noeud destinataire est fait comme dans le cas initial où tous les noeuds ont le même niveau de charge. Cela veut dire qu'une autre couche de processus sera placé sur les noeuds en augmentant leurs niveau de charge (si les noeuds ne sont pas saturés).

Sur la figure 5.31 nous pouvons voir, pour ce test, le nombre total de processus sur chaque noeud d'une grille de taille 8x8, selon la méthode de placement utilisée. Dans le cas de l'algorithme Aléatoire-4, les nombre de processus varie entre 0 et 87, dans l'algorithme du Gradient varie entre 5 et 46, finalement dans l'algorithme évolutif, les noeuds exécutent entre 1 et 28 processus.

En résumé, après avoir testé les algorithmes de placement sur plusieurs programmes du groupe 1-1, nous pouvons dire que pour ce premier plan d'expériences l'algorithme Evolutif a montré une bonne performance par rapport aux algorithmes du Gradient et l'aléatoire-4, en utilisant les plus de ressources de la machine et en équilibrant mieux la charge.

Méthode	*Paramètres*	*Machine*	*Temps*	*Nb.processus sur chaque noeud*
Aléatoire – 4	– – –	$m15x15$	76150	0 – 100
Gradient	*S. Moyen* : 2 *S. Chargé* : 3	$m15x15$	26368	0 – 23
Evolutif	*nb.Processus/NvCh* : 1 *Taille max SP* : 40 *Validité SP* : 50000	$m15x15$	33795	0 – 23
Aléatoire – 4	– – –	$m10x10$	80600	0 – 95
Gradient	*S. Moyen* : 2 *S. Chargé* : 3	$m10x10$	58292	0 – 47
Evolutif	*nb.Processus/NvCh* : 1 *Taille max SP* : 40 *Validité SP* : 50000	$m10x10$	35537	0 – 24
Aléatoire – 4	– – –	$m8x8$	64389	0 – 87
Gradient	*S. Moyen* : 2 *S. Chargé* : 3	$m8x8$	45549	5 – 46
Evolutif	*nb.Processus/NvCh* : 2 *Taille max SP* : 40 *Validité SP* : 50000	$m8x8$	39587	1 – 28

TAB. 5.1 - *Les temps d'exécution du programme "pparallel.pr" obtenus par les différentes méthodes de placement sur les grilles 15x15, 10x10 et 8x8*

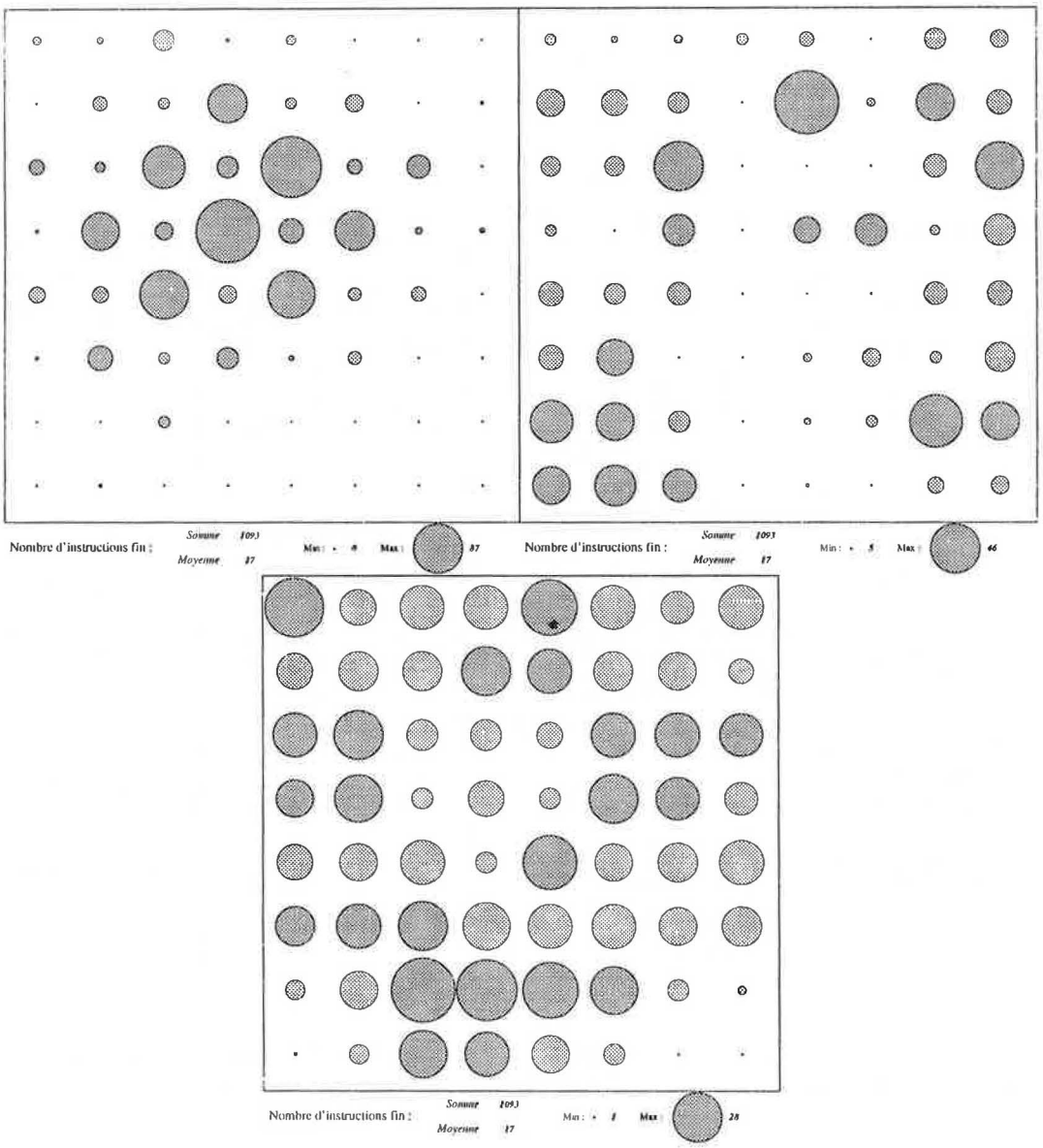


FIG. 5.31 - *Nombre de processus exécutés sur chaque processeur, Aléatoire-4,Aléatoire-25 (en haut), Gradient et Evolutif (en bas)*

5.3.2.3 Exécution des programmes avec d'autres graphes de créations

Dans cette partie nous allons étudier le comportement des algorithmes du Gradient, l'Aléatoire-4 et l'Evolutif sous l'exécution des programmes du groupe 1-2. Ces programmes de test génèrent peu de communications et ont de graphes de créations différents d'un arbre complet de hauteur $H > 1$. Nous présenterons le deuxième plan d'expériences utilisé dans cette étude.

Le plan d'expériences est formé par 35 programmes qui correspondent à un des 5 types de programme du groupe 1-2. Les 12 premiers programmes ont le type du programme 1 "p1pere-Nfils.prg", les programmes 13 au 24 ont le type du programme 2 "p1pere-Nfils-sync.prg", pour le type du programme 3 ("disques.prg") nous avons utilisé 5 programmes et pour chacun des deux derniers programmes "reines.prg" et "arbre-tableau.prg" nous utilisons 3 programmes. Dans la suite nous expliquerons les caractéristiques générales des programmes.

Pour les 12 premiers programmes du type 7, un seul processus initial crée N autres processus et chaque processus créé réalise une quantité de calculs dès qu'il est placé sur un noeud. A la fin de l'exécution des calculs chacun des fils envoi un dernier message de terminaison au processus initial. Pour les 12 programmes le volume mémoire des processus = 5000um, la quantité de calculs = 800000 et le dernier message envoyé au processus initial a un volume = 1000um. Les 12 programmes diffèrent par le nombre de processus créés qui est indiqué dans la table suivante:

Num. de programme	*Nb.Totalde processus*
1	21
2	41
3	61
4	81
5	101
6	151
7	201
8	301
9	401
10	601
11	801
12	1001

Le comportement des programmes 13-24 du type 8 ressemble un peu aux programmes du type 7 dans le sens que c'est un seul processus qui crée tous les autres. La différence est que les processus créés sont synchronisés avant de commencer leurs

calculs. Cela oblige à ce que tous les processus se trouvent placés sur la machine à un instant donné permettant la comparaison des distributions obtenues en utilisant les différents algorithmes de placement. Ces programmes ont les mêmes caractéristiques que les 12 premiers programmes par rapport au nombre de processus crées, le volume mémoire des processus = 5000um, la quantité de calculs = 800000 et le dernier message envoyé au processus initial a un volume = 2000um.

Les 5 programmes suivants du type 9 ("disqûes.prg") possèdent un graphe de créations en forme d'arbre irrégulier qui représente l'espace d'états dans la recherche de toutes les solutions du problème de la rotation de N disques. Chaque disque est divisé en 4 secteurs et dans chaque secteur il y a un numéro. Le problème est de faire tourner les N disques de telle sorte que la somme des numéros dans le même secteur de tous les disques soit inférieur ou égale à un numéro M. Chaque programme a un nombre différente de disques comme indiqué dans la table suivante:

Num. de programme	*Num.de Disques*	*Nb.Totalde processus*
25	6	260
26	7	401
27	8	586
28	9	821
29	10	1112

Pour les 3 programmes du type 10 ("reines.prg"), le graphe de créations est un arbre irrégulier qui représente l'espace d'états dans la recherche de toutes les solutions du problème de placer N reines sur un tableau de taille N. Chaque programme a une taille différente de tableau comme indiqué ci-dessous:

Num. de programme	*Taille du tableau*	*Nb.Total de processus*
30	6	150
31	7	513
32	8	1966

Les programmes 33 au 35 (du type 11 "arbre-tableau.prg") ont un graphe de créations en forme d'arbre mais le nombre de créations par processus dépend du niveau auquel ils appartiennent. Pour chaque programme il est défini un vecteur V de taille H-1 (où H est la hauteur de l'arbre) qui contient les nombre de créations à effectuer. La première valeur du vecteur indique le nombre de créations à effectuer par le processus racine, la deuxième valeur indique le nombre de créations à effectuer par les processus du niveau 2, etc. jusqu'à arriver à la dernière valeur qui indique le nombre de créations à effectuer par chaque processus au niveau H-1 de l'arbre.

Dans la liste suivante nous montrons les hauteurs des arbres associés à chacun des 3 programmes avec leurs vecteurs respectifs:

Num. de programme	*Hauteur de l'arbre*	*Vecteur avec le nb. de créations selon le niveau*	*Nb.Total de processus*
33	18	2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2	1533
34	15	2,1,3,1,2,1,3,1,2,1,3,1,2,1	1553
35	13	4,1,2,1,3,1,4,1,2,1,3,1	1801

En ce qui concerne les valeurs choisies pour les paramètres des algorithmes de placement, l'algorithme du Gradient a été initialisé comme suit: pour les programmes 1 à 18 les deux seuils ont été fixés à 1 et 2 processus, de cette façon les noeuds légèrement chargés (exécutant 0 ou 1 processus) peuvent accepter des processus externes, mais à partir d'un nombre de processus > 2 les processus créés localement doivent être placés sur d'autres noeuds. Pour les tests restants les seuils dans la méthode du Gradient ont pris les valeurs suivantes:

Num. de programme	*Seuils*
19	2 *et* 3
20	2 *et* 4
21	3 *et* 5
22	6 *et* 8
23	16 *et* 18
24	20 *et* 22
25 *au* 31	1 *et* 2
32	3 *et* 6
33 *au* 35	4 *et* 8

Les paramètres de l'algorithme Evolutif ont eu les valeurs suivantes: pour les tests 1 au 24 nous avons défini 1 processus par niveau de charge, la taille maximale des SP = 6 et le temps maximum de validité des SP = 500. Pour les tests qui restent (du 25 au 35) les paramètres sont: 1 processus par niveau de charge, la taille maximale des SP = 6 et le temps maximum de validité des SP = 50000ut.

La figure 5.32 montre le nombre maximum de noeuds utilisés à un moment donné selon les 3 algorithmes de placement, le Gradient, l'Aléatoire4 et l'Evolutif, exécutant ce deuxième plan d'expériences. On peut voir que pour les premiers programmes de type 7 "p1pere-Nfils.prg" (test 1-12) où un seul processus initial crée N processus, le nombre de noeuds utilisés n'est pas très important. Cela est du à ce que chaque processus créé réalise ses calculs sans attendre une synchronisation avec les autres processus créés. C'est à dire que le processus initial peut ne pas avoir fini de créer les

derniers processus lorsqu'une grande partie des processus déjà créés peut avoir fini leur exécution et avoir libéré les noeuds de la machine sur lesquels ils s'exécutaient. Pour ces premiers tests nous pourrions dire, de manière générale que l'algorithme Evolutif a utilisé le plus grand nombre de noeuds de la machine à un instant donné. Cependant il n'y a pas une grande différence par rapport aux noeuds utilisés avec les autres algorithmes.

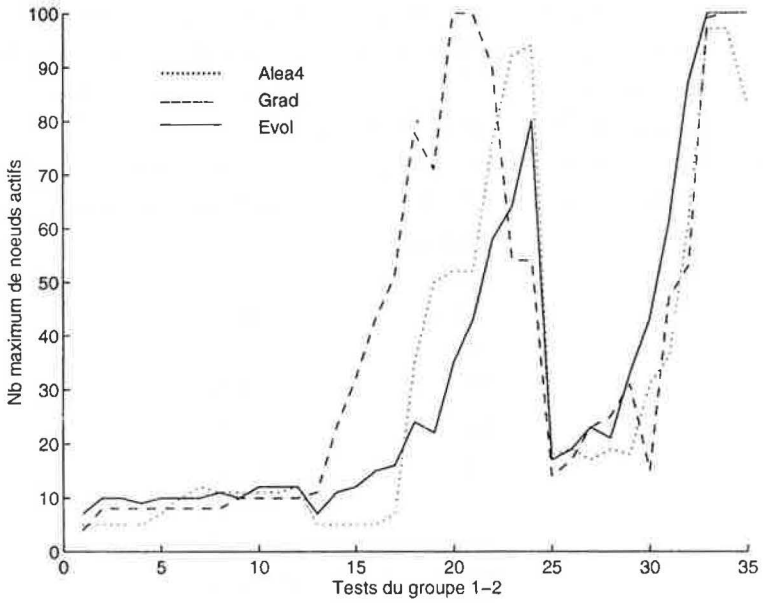


FIG. 5.32 - *Nombre maximum de noeuds utilisés à un moment donné*

Pour les programmes de type 8 "p1pere-Nfils-sync.prg" (les processus créés sont synchronisés avant d'exécuter leurs calculs. Ce comportement implique que à un moment donné tous les processus créés se trouveront placés sur les noeuds de la machine. Ce test nous a permis d'observer combien de noeuds on peut utiliser après une génération centralisée des processus. Nous pouvons voir que dans les test 16 au 24 la méthode du Gradient a pu utiliser le plus de noeuds de la machine, ensuite l'aléatoire et finalement l'algorithme Evolutif. Nous interprétons ce comportement de l'algorithme Evolutif comme le résultat du retard de l'arrivée des SP au noeud central de la machine, soit par les paramètres choisis ou bien par la surcharge des liens de communication.
Finalement pour les derniers tests (25-35) qui ont des graphes de création en forme d'arbres irréguliers, l'algorithme Evolutif a utilisé, d'une manière générale, le plus

de noeuds de la machine.

La figure 5.33 montre le nombre maximum de processus exécutés par un noeud. Dans ce cas les graphes montrent trois résultats différents: Pour les premiers tests (1-12), l'algorithme Aléatoire-4 a eu, de manière générale, le meilleur équilibrage de la charge sur les noeuds. A partir des test 13 au 24, c'est l'algorithme du Gradient qui a placé le moins de processus sur les noeuds. Finalement pour les test 24-35 (les programmes avec un graphe de créations irrégulier), l'algorithme Evolutif a mieux distribué les processus par rapport aux autres algorithmes.

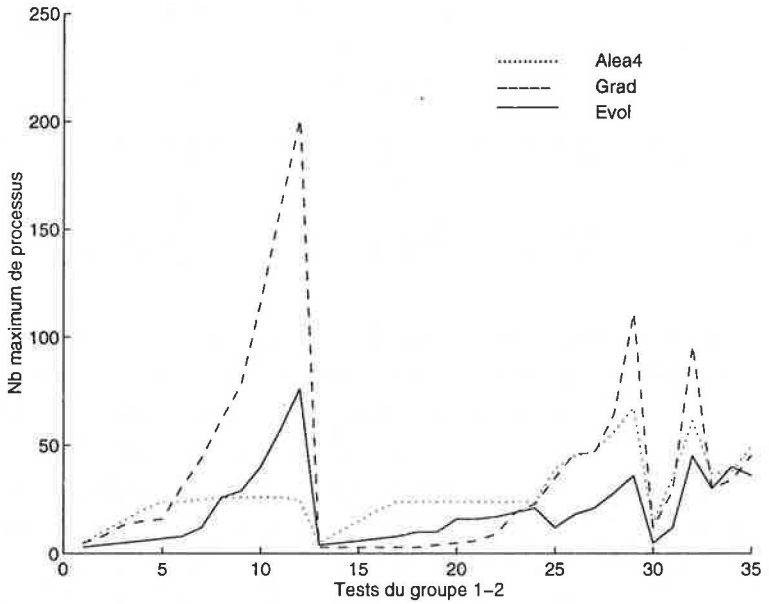


FIG. 5.33 - *Nombre maximum de processus exécutés par un noeud*

Finalement pour ce plan d'expériences nous avons sur la figure 5.34 les courbes qui montrent les temps d'exécution des programmes selon les différents algorithmes de placement. Lorsque les algorithmes exécutent les programmes du groupe 1-2, les temps d'exécution minimaux ont été obtenus par l'algorithme Evolutif et par l'Aléatoire-4. L'algorithme Evolutif a surtout obtenu les temps le plus petits dans l'exécution des programme avec un graphe de création irrégulier.

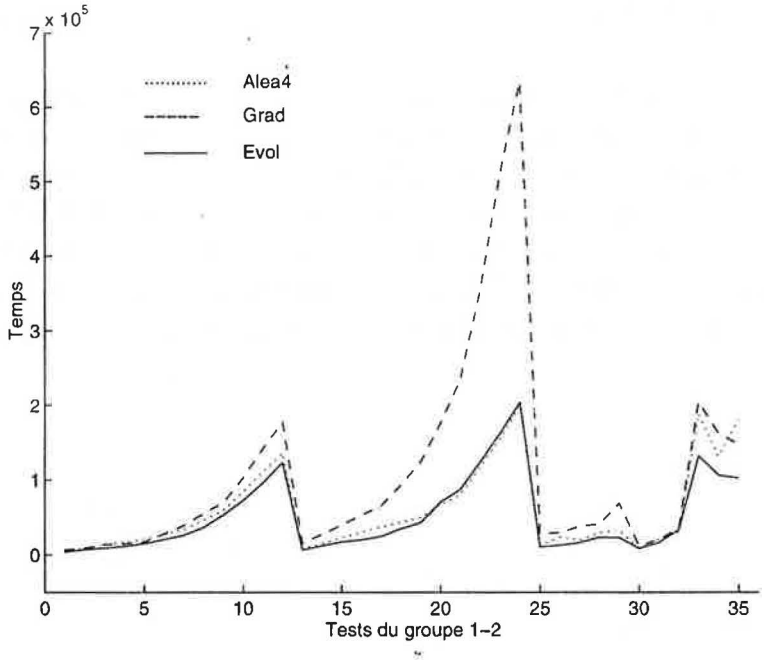


FIG. 5.34 - *Temps d'exécution des programmes du groupe 1-2*

De manière concrète et en comparant les trois graphes précédents, nous pouvons suggérer l'utilisation de certains algorithmes de placement pour l'exécution des programmes dans ce plan d'expériences. Pour les programmes de type 7 ou 8 où les créations sont générés par un seul processus central, l'utilisation de l'algorithme Evolutif ou bien de l'Aléatoire-4 donne les meilleurs résultats, car ils ont obtenu les temps d'exécution les plus petits, utilisent le plus de noeuds et distribuent plus équitablement les processus. La méthode du Gradient arrive a mieux utiliser les ressources de la machine mais malheureusement les temps d'exécution obtenus sont les plus grands.

Pour exécuter les programmes avec un graphe de création irrégulier où les processus ne génèrent pas beaucoup de communications l'algorithme Evolutif a obtenu les meilleures performances. En général il a utilisé le plus de noeuds de la machine à un instant donné, il a mieux distribué les processus sur les noeuds et il a obtenu les temps d'exécution les plus petits.

5.3.2.4 Un exemple particulier

Pour donner une comparaison plus détaillée des distributions des processus obtenues par les trois algorithmes de placement nous avons choisi un programme de type 11 "arbre-tableau.prg" (test No. 34) où l'algorithme Evolutif a donné une bonne dis-

tribution des processus sur les noeuds.

Sur le coin supérieur gauche de la figure 5.35 nous pouvons voir la distribution obtenue par la méthode du Gradient qui, dans ce test, a donné le plus grand temps d'exécution (258222ut). Comme on peut voir le nombre maximum de processus exécutés par chaque noeud varie entre 16 et 43 (27 processus de différence).

Dans ce test l'algorithme Aléatoire-4 a réduit le temps d'exécution du programme par rapport au temps donné par celui du Gradient (176189ut). Nous pouvons voir sur le coin supérieur droit de la figure 5.35 la distribution des processus obtenue par cet algorithme. La différence entre le plus grand nombre et le plus petit nombre de processus exécutés par un noeud est plus grande que dans le cas du Gradient (41-8=33 processus de différence).

En ce qui concerne l'algorithme Evolutif, dans ce test il a donné le temps d'exécution le plus petit (135275ut). Dans la partie basse de la figure 5.35 nous avons la distribution obtenue des processus sur la grille utilisant cet algorithme. Comme nous pouvons voir, la différence entre le nombre minimum et le nombre maximum des processus exécutés par un noeud est la plus petite que dans les cas de l'Aléatoire-4 et celui du Gradient (34 - 12 = 22).

Nous avons présenté une comparaison des 3 algorithmes de placement exécutant les programmes de test du groupe 1 où les processus communiquent peu. Dans la suite nous allons comparer l'algorithme Evolutif avec l'Aléatoire-4 et celui du Gradient exécutant des programmes où les processus réalisent plusieurs échanges de messages entre eux.

5.3.3 Comparaison sur le groupe 2 de programmes où les processus font plusieurs échanges de messages

Cette partie est consacrée à la comparaison des trois algorithmes de placement pour l'exécution des programmes du groupe 2 qui génèrent plus de messages sur la machine que ceux du groupe 1. Nous commencerons d'abord par la comparaison des algorithmes exécutant les plans d'expériences formés par les programmes qui ont un graphe de créations en forme d'arbre complet (groupe 2-1). Ensuite nous présenterons la dernière comparaison obtenue en exécutant les programmes qui provoquent plusieurs échanges de messages et qui ont des graphes de créations différents d'arbres complets.

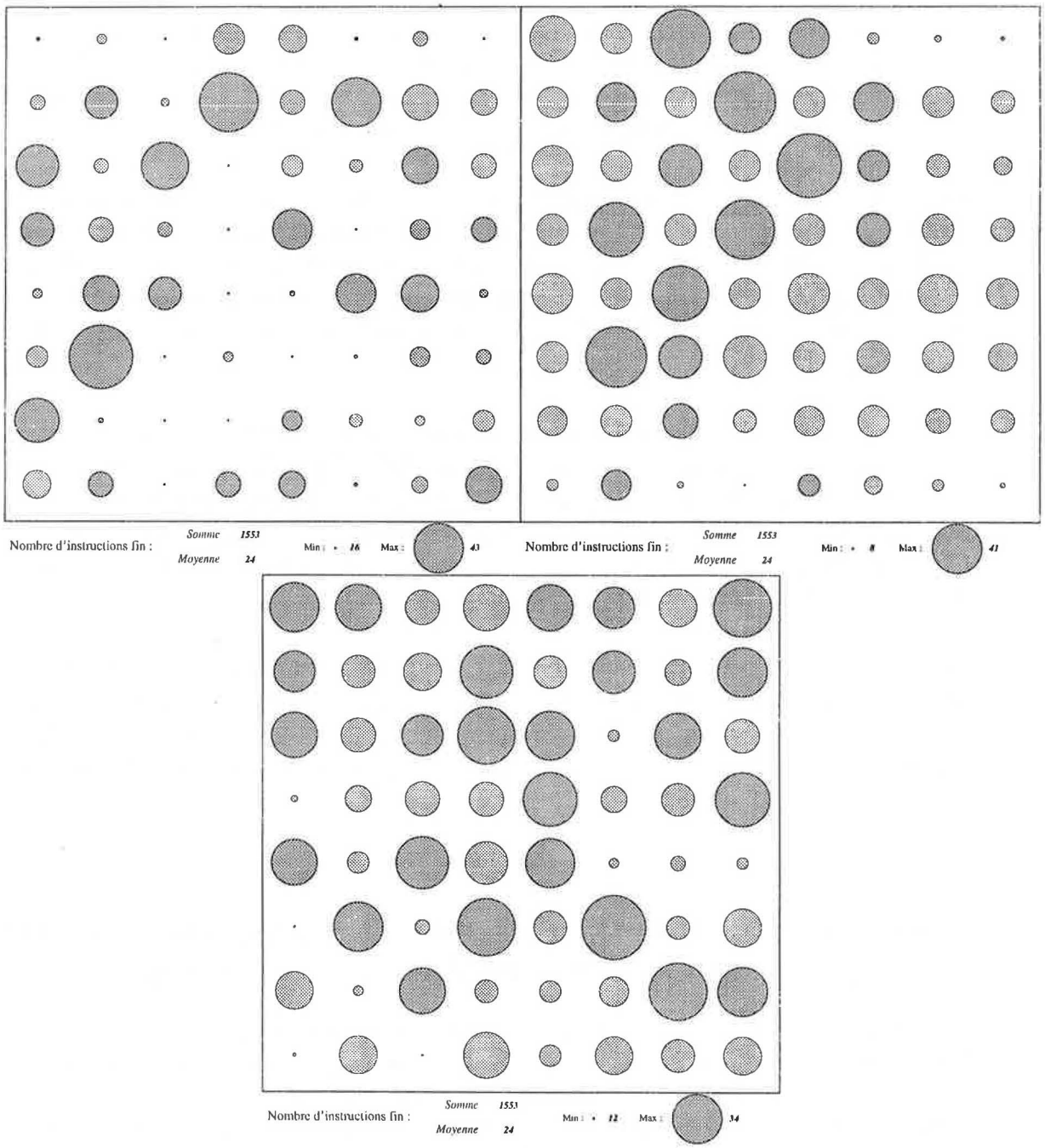


FIG. 5.35 - *Nombre de processus exécutés sur chaque processeur, Aléatoire-4,Aléatoire-25 (en haut), Gradient et Evolutif (en bas)*

5.3.3.1 Exécution des programmes avec un graphe de créations en forme d'arbre complet

La comparaison des algorithmes de placement exécutant les programmes du groupe 2-1 est divisée en 3 parties. Chacune des parties utilise un plan d'expériences formé par plusieurs programmes de même type.

Le premier plan d'expériences est formé par 44 programmes du type de programme 12 ("arbre-complet-2.prg"). Dans ce cas, les processus fils font N échanges de messages avec leurs pères (1 échange = 2 envois de messages), et entre chaque envoi et réception il font une même quantité de calculs. Une fois que les échanges et les calculs sont finis, les processus fils envoient un dernier message de terminaison à leurs pères.

Les caractéristiques communes des 44 programmes sont: volume mémoire du processus initial = 2000um, volume mémoire des processus de l'arbre = 5000um, quantité de calculs = 10000uc et volume des messages échangés = 8000um. Les 44 programmes diffèrent par le nombre de processus créés et le nombre d'échanges effectués entre chaque père et fils comme indiqué ci-dessous

Num. de programme	*Nb.Créat par processus*	*Hauteur de l'arbre*	*Nb.Total de processus*	*Nombre d'échanges*
1 *au* 4	2	9	512	1, 2, 4 *et* 6
5 *au* 8	3	6	365	1, 2, 4 *et* 6
9 *au* 12	4	5	342	1, 2, 4 *et* 6
13 *au* 16	5	4	157	1, 2, 4 *et* 6
17 *au* 20	6	4	260	1, 2, 4 *et* 6
21 *au* 24	7	4	401	1, 2, 4 *et* 6
25 *au* 28	8	3	74	1, 2, 4 *et* 6
29 *au* 32	9	3	92	1, 2, 4 *et* 6
33 *au* 36	10	3	112	1, 2, 4 *et* 6
37 *au* 40	15	3	242	1, 2, 4 *et* 6
41 *au* 44	20	3	422	1, 2, 4 *et* 6

Pour le deuxième plan d'expériences nous utilisons 44 programmes (du 45 au 88) du type du programme 13 ("prg-comm-1.prg"). Le comportement de ces programmes resemble à celui des premiers programmes, chaque processus fils fait N échanges de messages avec son père et entre chaque envoi et reception fait une quantité de calculs. La différence est que dans ces programmes les processus fils font une quantité de calcul égale à la quantité de calcul faite par leurs pères moins une quantité M de calcul.

Les caractéristiques communes des programmes 45 au 88 sont: volume mémoire des processus = 5000um, volume des messages envoyés = 1500um, la quantité de cal-

culs effectuée par le processus racine = 800000 et chaque processus fils réalise la même quantité de calculs que son père moins 40000uc. Les programmes 45 au 88 se différencient par le nombre de processus créés et par le nombre d'échanges entre processus comme dans les 44 premiers programmes.

Le dernier plan d'expériences du groupe 2-1, contient 44 programmes (du 89 au 132) du type de programme 14 ("comm3FFH7.prg"). Dans ces programmes chaque processus fils fait N échanges de messages avec chacun de ses frères et entre chaque envoi et réception fait une quantité de calculs. Les placements trop distants de deux processus frères peuvent augmenter les temps d'exécution résultants.

Les caractéristiques communes des programmes dans le troisième plan d'expériences sont: volume mémoire du processus initial = 1000um, volume mémoire des processus dans l'arbre = 5000um et volume des messages envoyés = 80000um. La quantité de calculs effectués par le processus racine = 150000, pour les autres processus, il font la même quantité de calculs faite par leurs pères moins 12000uc. De la même manière que les programmes du type 1, les programmes 89 au 132 diffèrent par le nombre de processus créés et par le nombre d'échanges effectués.

Pour exécuter les trois plans d'expériences nous avons initialisé les paramètres des algorithme de placement comme suit: L'algorithme du Gradient a eu les seuils 2 et 6, permettant à chaque noeud d'accepter jusqu'à 2 processus externes et de distribuer les processus locaux lorsque sa charge est plus grande que 6 processus.
En ce qui concerne l'algorithme Evolutif ses paramètres ont eu les valeurs suivantes: 1 processus par niveau de charge, une taille maximale des SP = 6 et le temps maximum de validité des SP = 50000.

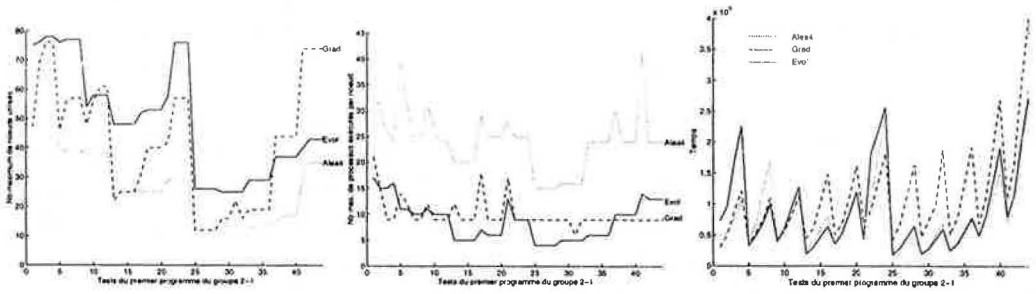


FIG. 5.36 - *Temps d'exécution, nombre maximum de noeuds actifs et nombre maximum de processus exécutés par noeud pour le programme 1 du groupe 2-1*

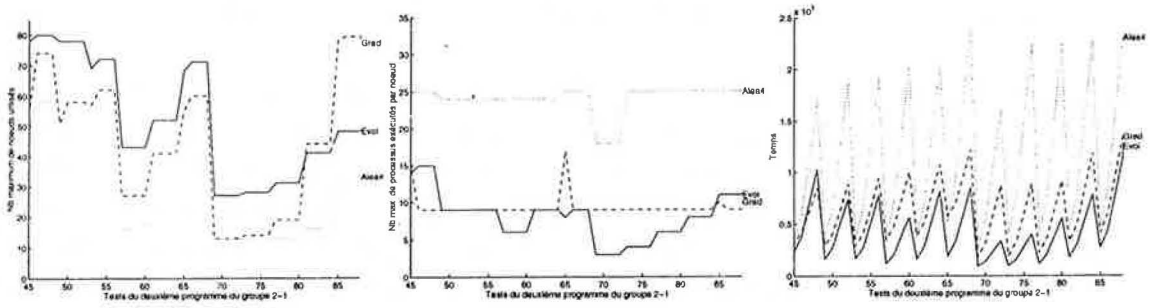


FIG. 5.37 - *Temps d'exécution, nombre maximum de noeuds actifs et nombre maximum de processus exécutés par noeud pour le programme 2 du groupe 2-1*

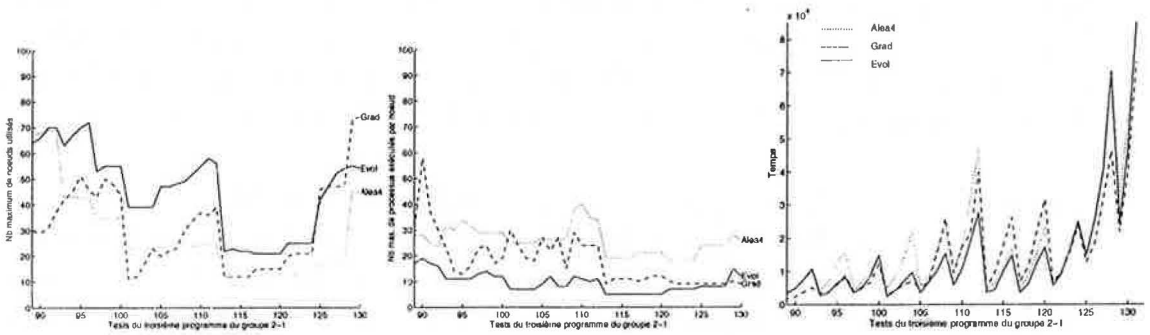


FIG. 5.38 - *Temps d'exécution, nombre maximum de noeuds actifs et nombre maximum de processus exécutés par noeud pour le programme 3 du groupe 2-1*

Les graphes des figures 5.36, 5.37 et 5.38 montrent: le nombre de noeuds utilisés en même temps à un instant donné, le nombre maximum de processus exécutés par un noeud et les temps d'exécution obtenus par les différents algorithmes de placement pour les 3 programmes du groupe 2-1.

On peut voir que lorsque la grille 10x10 exécute les deux premiers plans d'expériences le nombre maximum de noeuds utilisés à un instant donné varie presque de la même façon (graphes à gauche des figures 5.36 et 5.37). Pour les 36 premiers tests des deux plans d'expériences, l'algorithme Evolutif obtient les plus grands nombres de noeuds utilisés, ensuite l'algorithme du Gradient et finalement l'Aléatoire-4. Pour les tests 37 au 44 et 81 au 88 (où les processus font le plus grand nombre de créations) c'est l'algorithme du Gradient qui a utilisé le plus de noeuds de la grille, en

deuxième place se trouve l'algorithme Evolutif et à la fin l'Aléatoire-4.

Dans le troisième plan d'expériences, les nombres maximaux de noeuds utilisés selon chacun des algorithmes de placement change un peu. De manière générale c'est l'algorithme Evolutif qui a donné les meilleurs résultats, seulement pour les derniers tests (125-132) il a eu la deuxième place après l'algorithme du Gradient. Nous pouvons voir que dans l'exécution du troisième plan d'expériences l'algorithme Aléatoire-4 a pu améliorer à l'algorithme du Gradient dans quelques tests comme c'est montré sur le graphe à gauche de la figure 5.38.

De manière générale, nous avons trouvé que dans les trois plans d'expériences l'algorithme Evolutif a obtenu les plus petit nombres maximaux de processus exécutés par un noeud, comme indiqué sur les graphe du centre des figures 5.36, 5.37 et 5.38. En ce qui concerne les autres algorithmes de placement, la méthode du Gradient a eu la deuxième place donnant des valeurs proches de celles données par l'algorithme Evolutif (surtout dans les deux premiers plan d'expériences). C'est l'algorithme Aléatoire-4 qui a eu la dernière place donnant, de manière générale, les plus grand nombres maximaux de processus exécutés par un noeud.

Nous avons à la droite des figures 5.36, 5.37 et 5.38 les temps d'exécution obtenus par chaque algorithme de placement dans les trois plans d'expériences respectivement. Pour le premier plan d'expériences où les processus font tous une même quantité de calculs entre chaque échange de messages, l'algorithme Evolutif et l'Aléatoire-4 ont obtenu, de manière générale, les plus petits temps d'exécution. Seulement dans certains test (du 1 au 4 et du 21 au 24) le Gradient a pu améliorer les temps d'exécution.

En exécutant les programmes du deuxième plan d'expériences nous avons remarqué une amélioration des temps d'exécution obtenus par l'algorithme Evolutif et la méthode du Gradient par rapport à ceux obtenus par l'Aléatoire-4 (comme montré sur la figure à la droite de la figure 5.37). Dans la plus grande partie des test l'algorithme Evolutif a obtenu les plus petits temps d'exécution des programmes et la méthode du Gradient a eu la deuxième place . Dans ce plan d'expériences où les processus fils font moins de calculs que leurs pères entre chaque échange de messages, l'algorithme Aléatoire-4 a eu la dernière place donnant le plus grands temps d'exécution.

Finalement, pour le troisième plan d'expériences où les processus font des échanges de messages entre frères, nous avons trouvé que dans la plus grande partie des tests les plus petits temps d'exécution ont été obtenus par l'algorithme Evolutif. Nous pouvons voir sur les courbes à droite de la figure 5.38 que seulement dans quelques tests la méthode du Gradient a amélioré les temps obtenus par l'algorithme Evolutif.

D'autre part, l'algorithme Aléatoire-4 a donné des temps d'exécution égaux ou plus grands que l'algorithme Evolutif.

Après avoir comparé les algorithmes de placement sous l'exécution des programmes de test du groupe 2-1 nous avons remarqué que l'algorithme Aléatoire-4 ainsi que l'Evolutif ont eu un bon comportement exécutant des programmes du type 12 ("arbre-complet-2.prg") où les processus pères et fils font une même quantité de calculs entre chaque échange de messages. Lorsqu'on exécute des programmes du type 13 ("prg-comm-1.prg") où les processus pères font plus de calculs que les processus fils entre chaque échange de messages, les algorithmes Evolutif et du Gradient peuvent être utilisés pour obtenir des bonnes performances par rapport aux trois facteurs: le nombre de noeuds utilisés, l'équilibrage de la charge et le temps d'exécution obtenu. En prenant en compte ces trois facteurs, l'algorithme Evolutif a montré des bonnes performances en exécutant les programmes du type 14 ("comm3FFH7.prg"). En résumé nous pouvons dire que l'algorithme Evolutif a donné des bons résultats en exécutant les 3 programmes de test du groupe 2-1 où les processus font plusieurs échanges de messages et ont un graphe de création en forme d'arbre complet.

5.3.3.2 Un exemple particulier

Pour donner une comparaison plus détaillée des algorithmes de placement exécutant des programmes du groupe 2-1, nous avons utilisé deux programme du type 14 ("comm3FFH7.prg") pour les faire exécuter sur une grille de taille 8x8. Dans ces programmes les processus frères font M échanges de messages et font une même quantité de calculs entre chaque échange.

L'objectif de ce test est d'observer les comportements des méthodes lorsque l'évolution de la charge sur la machine est provoquée par des programmes indépendants. L'élément d'information de chacune des méthodes de placement doit être capable de s'adapter à de tels changements pour réguler la charge des processeurs.

Le processus initial du premier programme a été placé sur le noeud du coin supérieur gauche de la grille tandis que le processus initial du deuxième programme a été placé sur le noeud du coin inférieur droit.

Pour les deux programmes le nombre de créations par processus est égale à 3 et la hauteur de l'arbre est égale à 6. Le nombre total de processus à placer est donc 365+365 = 730. Les variables des programme ont été initialisées comme suit: quantity_comp = 150000, vol_comm = 8000, son_number = 3, tree_height = 6, nb_comm_frere = 3 et pour chaque niveau de l'arbre la quantité de calculs à faire

par les processus décroît de 12000 uc. L'initialisation des paramètres et les temps d'exécution obtenus utilisant les algorithmes de placement Evolutif, du Gradient, Aléatoire-4 et Aléatoire-25 sont montrés dans la table suivante:

Méthode	*Paramètres*	*Temps*	*Nb.processus sur chaque noeud*
Aléatoire − 25	− − −	172532	0 − 39
Aléatoire − 4	− − −	97011	0 − 54
Gradient	*S. Moyen* : 4 *S. Chargé* : 6	96079	4 − 23
Evolutif	*nb.Processus/NvCh* : 1 *Taille max SP* : 4 *Validité SP* : 500000	59391	6 − 19

Comme nous pouvons remarquer, l'algorithme Evolutif a obtenu les temps d'exécution les plus petits dans ce test, ensuite la méthode du Gradient et les deux méthodes Aléatoire-4 et Aléatoire-25. Les graphes de la figure 5.39 montrent les nombres maximaux de processus exécutés par chaque noeud selon les différentes méthodes de placement.

Sur le coin supérieur gauche de la figure 5.39 nous avons la distribution des processus obtenue en utilisant l'algorithme Aléatoire-25. Dans ce cas il y a eu des noeuds qui n'ont pas été utilisés dans l'exécution, car le nombre de processus exécutés par un noeud varie entre 0 et 39. Cette distribution des processus a provoqué la sur-utilisation de certains liens de communication comme indiqué sur le graphe qui est au coin supérieur gauche de la figure 5.40. Le temps d'utilisation des liens de la machine a varié entre 2182 et 161068 ut.

Nous avons sur le coin supérieur droit de la figure 5.39 la distribution des processus obtenue en utilisant l'algorithme aléatoire-4. Cet algorithme a pu diminuer le temps d'exécution obtenu par l'Aléatoire-25 mais le prix payé a été la diminution aussi des noeuds utilisés dan l'exécution. Dans ce cas chacun des noeuds a exécuté entre 0 et 54 processus ayant provoque une agglomération surtout vers les coins supérieur gauche et inférieur droit de la grille. L'agglomération a généré aussi la sur-utilisation des liens qui sont vers ces deux coins donnant une utilisation entre 0 et 94284 ut (comme indiqué sur le graphe du coin droit de la figure 5.40).

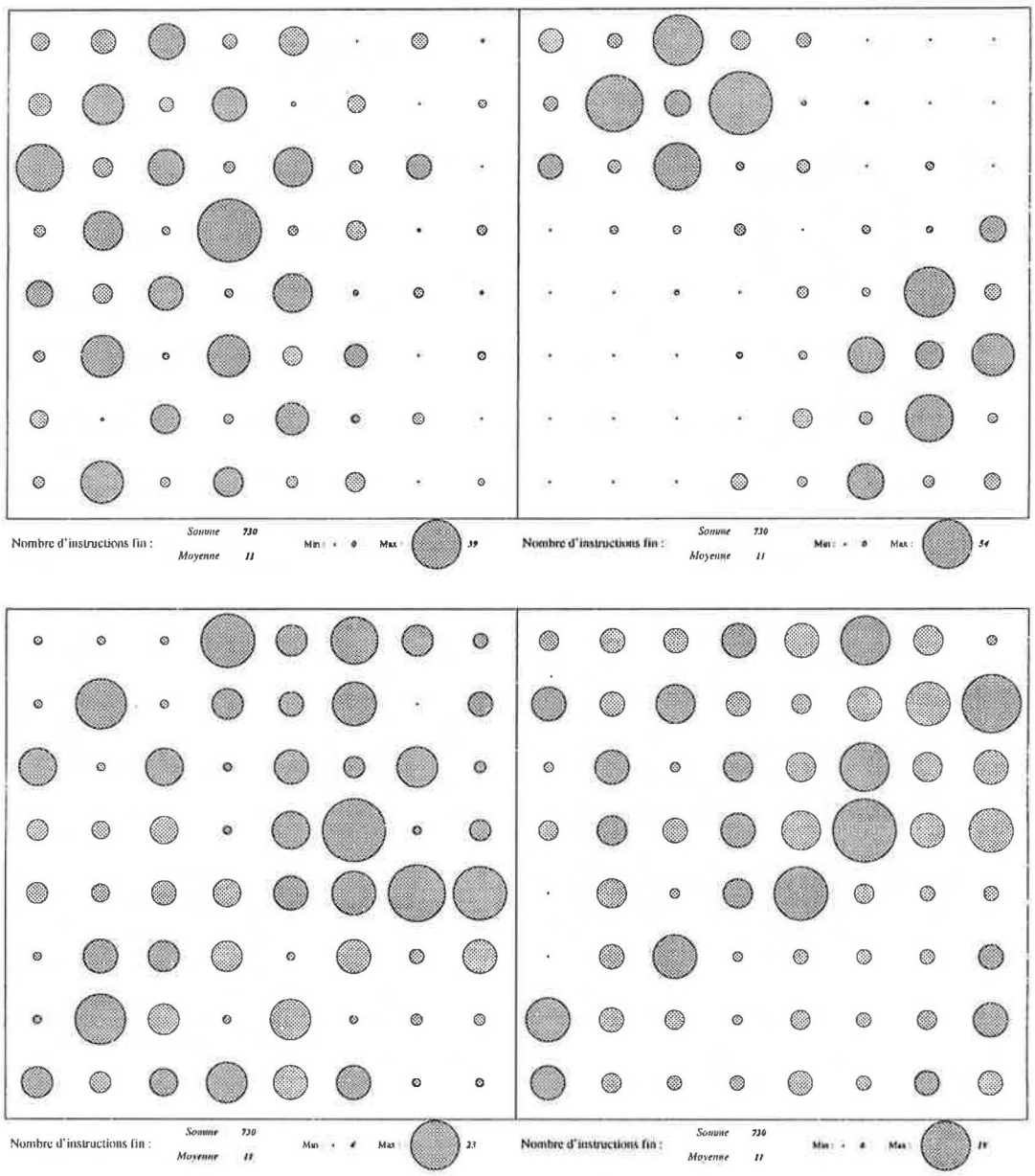


FIG. 5.39 - *Nombre de processus exécutés sur chaque processeur. Aléatoire-4, Aléatoire-25 (en haut), Gradient et Evolutif (en bas)*

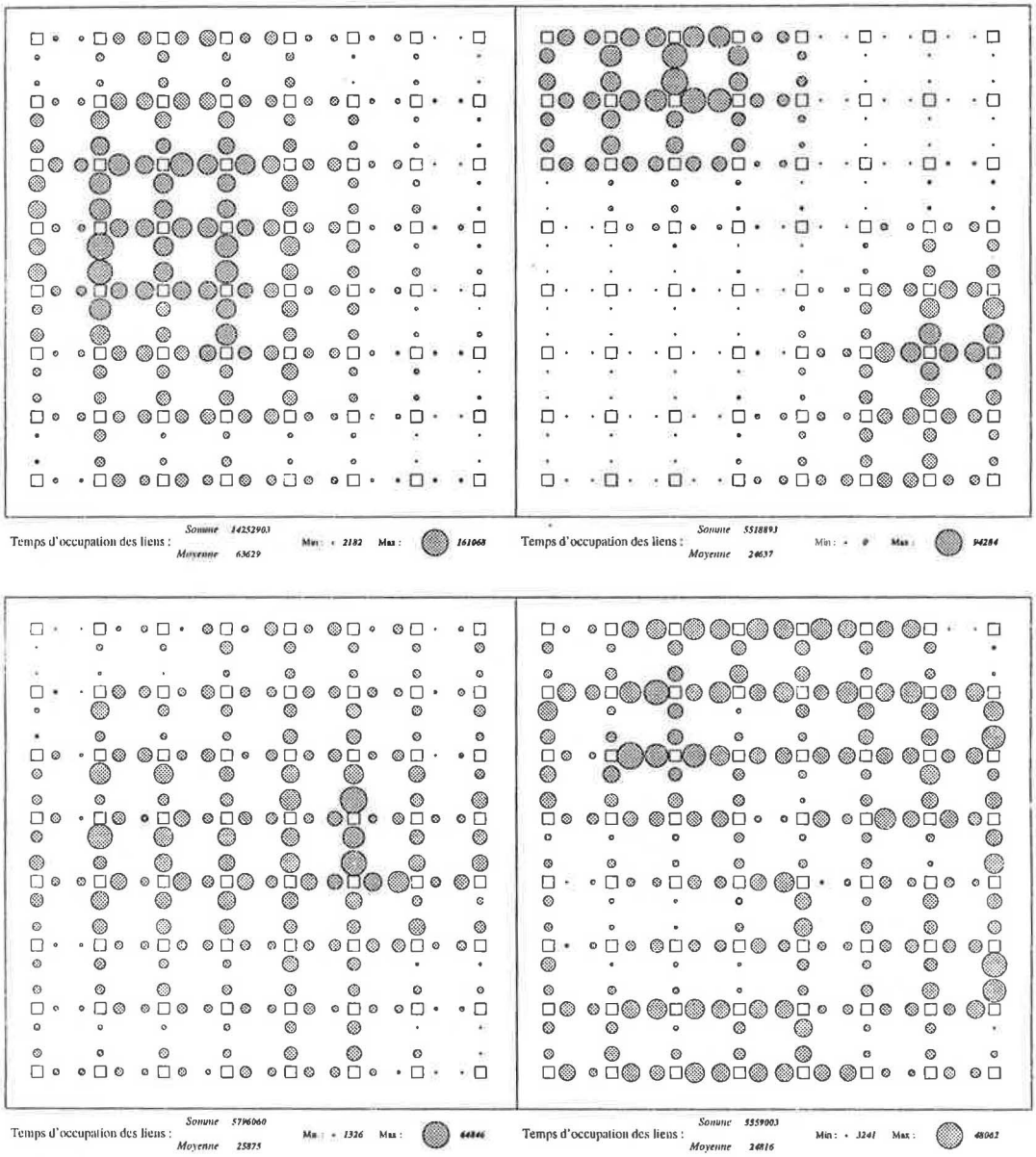


FIG. 5.40 - *Temps d'occupation des liens. Aléatoire-4, Aléatoire-25 (en haut), Gradient et Evolutif (en bas)*

L'autre méthode qui a encore amélioré le temps d'exécution de ce test a été celle du Gradient. Sur le coin inférieur gauche de la figure 5.39 nous avons la distribution des processus obtenue avec cette méthode. Dans ce cas tous les noeuds ont été utilisés et ils ont exécutés entre 4 et 23 processus. Par rapport aux liens de la machine, ils ont été utilisés plus équitablement ayant un temps d'utilisation qui varie entre 1326 et 64846 ut (comme indiqué sur le coin inférieur gauche de la figure 5.40).

Finalement, sur le coin inférieur droit de la figure 5.39 nous avons la distribution des processus obtenu en utilisant l'algorithme Evolutif, qui a donné le temps d'exécution le plus petit de ce test. Dans ce cas l'algorithme Evolutif a encore amélioré la distribution obtenue par la méthode du Gradient donnant à chaque noeud entre 6 et 19 processus. De la même manière l'utilisation des liens de communication a été mieux équilibrée variant entre 3241 et 48062 ut comme indiqué sur le graphe inférieure droite de la figure 5.40.
En général on peut voir que dans ce test l'algorithme Evolutif a utilisé les noeuds et les liens de la machine, d'une manière plus distribuée et équitable par rapport aux autres méthodes.

5.3.3.3 Les programmes avec d'autres graphes de créations

Cette partie montre une comparaison des algorithmes de placement exécutant le plan d'expériences formé par 36 programmes qui sont d'un des 4 types de programmes du groupe 2-2. Les programmes ont des processus qui font plusieurs échanges de messages et qui ont des graphes de créations différents aux arbres complets de hauteur $H > 2$.

Les 8 premiers programmes sont du type du programme 17 ("espace-d-etats.prg") qui a un graphe de créations en forme d'arbre irrégulier. L'arbre de créations représente l'espace d'états dans la recherche d'une solution pour le problème de placer N reines sur un tableau de taille NxN. Dans ce cas les processus créés font un certain nombre d'échanges avec leurs pères et entre chaque envoi et réception les processus font une certaine quantité de calculs.

Les caractéristiques communes des 8 premiers programmes sont: le volume mémoire pour les processus de l'arbre = 5000um, la quantité de calculs effectués par le processus racine = 250000uc, les processus fils effectuent une quantité de calculs égale à la quantité faite par son père moins 12000uc. Les processus qui sont les feuilles de l'arbre effectuent 5000uc et le volume mémoire des messages envoyés est égale à 2500um.

Les programmes diffèrent par le nombre de processus créés et par le nombre d'échanges réalisés comme indiqué ci-dessous

Num. de programme	*Taille du tableau*	*Nb.Total de processus*	*Nombre d'échanges*
1	6	150	1
2	6	150	2
3	6	150	4
4	6	150	6
5	7	513	1
6	7	513	2
7	7	513	4
8	7	513	6

En ce qui concerne l'initialisation des paramètres qui définissent les algorithmes de placement, pour ces 8 premiers programmes, l'algorithme Evolutif a été initialisé comme suit: 1 processus par niveau de charge, une tailles maximale des SP = 6 et un temps maximal de validité des SP = 5000ut. D'autre part la méthode du Gradient a eu son seuil moyen = 3 processus et son seuil chargé = 5 processus.

Les programmes 9 au 16 (du type "fourier.prg") ont un comportement où un seul processus crée d'autres N processus (où $N = 2^k$, $k >= 1$). Les processus font plusieurs échanges de messages comme dans le graphe de communications qui s'utilise pour résoudre la transformée rapide de fourier en parallèle. Une fois que les N processus ont été placés le processus initial prend l'initiative et envoie aux N processus un vecteur contenant les identificateurs respectifs pour faire les échanges.

Les caractéristiques communes de ces programmes du type "fourier.prg" sont: un volume mémoire du processus initial = 30000um, le volume mémoire des N processus = 8000um et une quantité de calculs effectué entre chaque envoi et réception de messages = 15000uc. Les 8 programmes diffèrent par le nombre de processus créés et par le volume du message envoyé dans les échanges, comme indiqué ci-dessous

Num. de programme	*Nb.Total de processus*	*Volume communications*	*Seuils du Gradient*
9, 17	65	3000	$m : 1ch : 2$
10, 18	65	80	$m : 1ch : 2$
11, 19	129	3000	$m : 2ch : 3$
12, 20	129	80	$m : 2ch : 3$
13, 21	257	3000	$m : 3ch : 4$
14, 22	257	80	$m : 3ch : 4$
15, 23	513	3000	$m : 6ch : 8$
16, 24	513	80	$m : 6ch : 8$

Sur la liste ci-dessus nous avons aussi écrit, pour chaque programme du type "fourier.prg", les valeurs avec lesquelles les seuils de la méthode du Gradient ont été initialisés (m= seuil moyen et ch= seuil chargé). En ce qui concerne les paramètres de l'algorithme Evolutif leur initialisation a été la suivante: 1 processus par niveau de charge, une taille maximale des SP=6 et un temps maximum de validité des SP = 80ut.

Les programmes 17 au 24 du type "fourier-par.prg" ont les mêmes caractéristiques que les programmes antérieurs (également pour l'initialisation des paramètres des algorithmes de placement). La différence est que ce sont les N processus créés qui prennent l'initiative de demander au processus initial les identificateurs avec lesquels faire des échanges. Cela augmente le nombre de messages dans la machine.

Finalement les programmes 25 au 36 sont du type du programme 18 ("prodN-consM.prg") où plusieurs systèmes, qui associent 1 consommateur avec N producteurs et 1 producteur avec N consommateurs, sont exécutés. Dans chacun des programme les nombre de systèmes et le volume du produit varient. Les systèmes associent 1 consommateur avec 10 producteurs et 1 autre producteur avec 10 autres consommateurs. Le consommateur associé aux 10 producteurs consomme 30 produits (chacun des producteurs produit 3 produits) et le producteur associé aux 10 consommateurs produit 30 produits (chacun des consommateurs consomme 3 produits).

Les variables du programme ont été initialisées comme suit: volume mémoire du processus initial = 2000um, les volumes mémoire des processus qui associent 1 producteur à N consommateurs et 1 consommateur à N producteurs = 6000um, le volume mémoire du processus consommateur et du processus producteur = 8000um. Dans le cas où un consommateur est associé a 10 producteurs, le temps de consommation = 10000ut et les temps de production = 10000*Nut. D'autre part, si un seul producteur est associé à N consommateurs, le temps de production = 10000ut et les temps de consommation = 10000*Nut.

La liste ci-dessous montre les programmes utilisés selon le nombre de systèmes créés, le volume du produit dans chaque programme et les valeurs des paramètres utilisés

par la méthode du Gradient.

Num. de programme	*Nb.de systèmes*	*Nb.Total de processus*	*Volume du produit*	*Seuils du Gradient*
25	5	126	50	$m:3\ ch:7$
26	5	126	500	$m:3\ ch:7$
27	10	251	50	$m:4\ ch:10$
28	10	251	500	$m:4\ ch:10$
29	15	376	50	$m:5\ ch:9$
30	15	376	500	$m:5\ ch:9$
31	20	501	50	$m:10\ ch:15$
32	20	501	500	$m:10\ ch:15$
33	25	626	50	$m:10\ ch:15$
34	25	626	500	$m:10\ ch:15$
35	30	751	50	$m:12\ ch:18$
36	30	751	500	$m:12\ ch:18$

Pour ces derniers tests les paramètres de l'algorithme Evolutif ont été initialisés comme suit: 1 processus par niveau de charge, une taille maximale des SP = 6 et un temps maximum de validité des SP = 5000ut.

Sur la figure 5.41 nous avons les courbes qui montrent le nombre maximum de noeuds utilisés dans un moment donné, selon les différents algorithmes exécutant ce dernier plan d'expériences. A première vue nous pouvons dire que pour les programmes du type "espace-d-etats.prg" l'algorithme Evolutif et celui du Gradient ont mieux profité des ressources de la machine par rapport à l'algorithme Aléatoire-4. Pour les programmes du type "fourier.prg" et "fourier-par.prg" ce sont les algorithmes Aléatoire-4 et celui du Gradient qui ont utilisé le plus de noeuds de la machine que l'algorithme Evolutif. Dans ce cas où un seul noeud centralisé crée tous les processus du programme, l'algorithme Evolutif met plus de temps à actualiser la SP du noeud central.
Finalement, dans l'exécution des derniers programmes du type "prodN-consM.prg" (test 25 au 35) où les créations sont à nouveau distribuées l'algorithme Evolutif a utilisé le plus de noeuds de la machine.

La figure 5.42 montre les nombres maximaux de processus exécutés par un noeud obtenus par les trois algorithmes de placement dans ce plan d'expériences. Nous pouvons voir que pour les tests 1 au 24, l'algorithme du Gradient a fait exécuter le moins de processus sur les noeuds de la machine. En deuxième place se trouve l'algorithme Evolutif et à la fin l'Aléatoire-4. Pour la deuxième partie des programmes

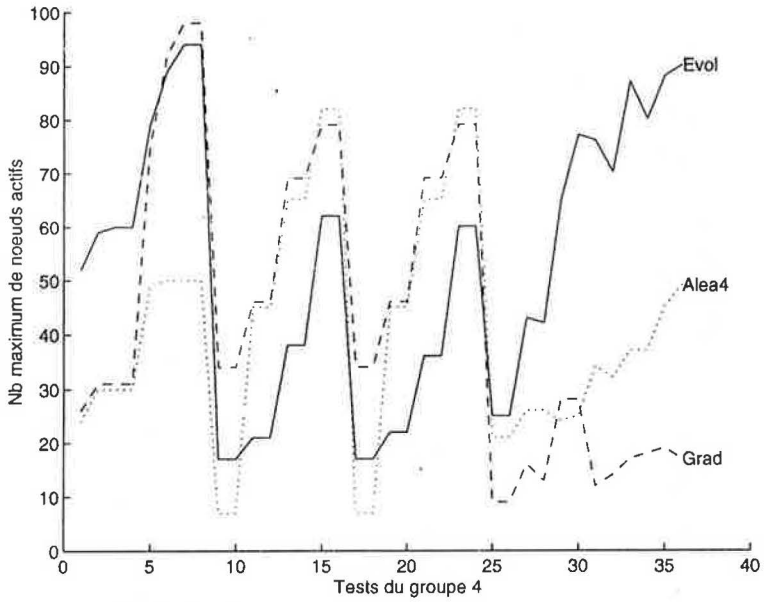


FIG. 5.41 - *Nombre maximum de noeuds actifs dans un instant donné*

c'est l'algorithme Evolutif qui a mieux équilibré la charge par rapport aux autres deux méthodes.

Finalement sur la figure 5.43 nous avons les courbes qui montrent les temps d'exécution obtenus par les trois algorithmes de placement. De manière générale, nous pouvons voir que l'algorithme Evolutif et l'Aléatoire ont donné les temps d'exécution les plus petits dans tous les tests.

En résumé nous pouvons dire que l'algorithme Evolutif a été capable d'exécuter les programmes du groupe 2-2 avec des bons résultats. Même dans le cas où les créations sont centralisées et malgré qu'il utilise un peu moins de noeuds que les méthodes du Gradient et l'Aléatoire-4, l'algorithme Evolutif est un des algorithmes qui obtient les plus petits temps d'exécutions (aussi l'Aléatoire-4) et donne un équilibrage de la charge meilleur que celui donné par l'algorithme Aléatoire-4.

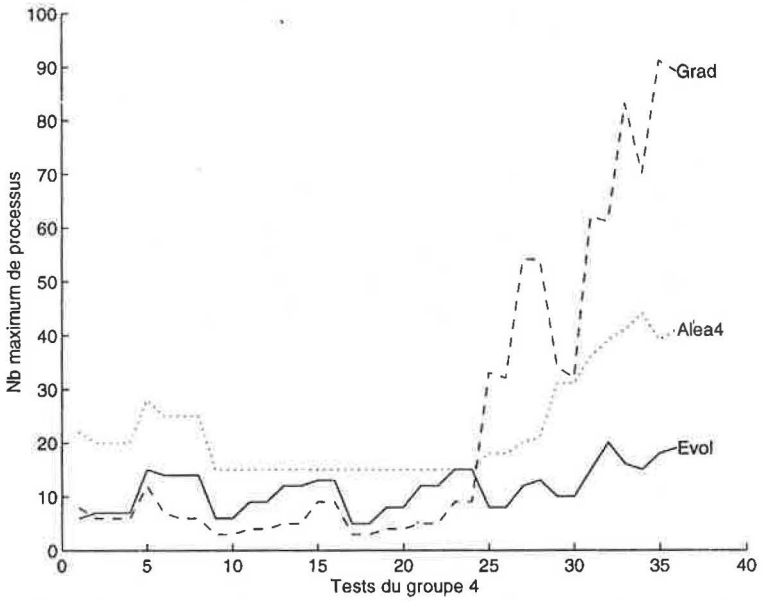


FIG. 5.42 - *Nombre maximum de processus exécutés par un noeud*

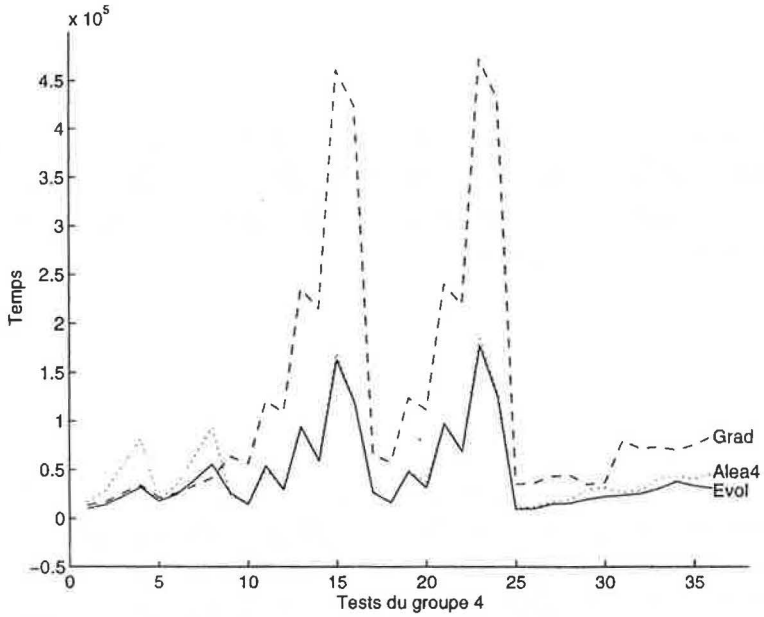


FIG. 5.43 - *Temps d'exécution des programmes du groupe 2-2*

5.3.3.4 Quelques exemples particuliers

Dans cette partie nous allons présenter deux comparaisons particulières des algorithmes de placement exécutant des programmes du groupe 2-2. La première comparaison utilise le programme 23 du plan d'expériences du groupe 2-2 pour montrer les distributions des processus obtenues par les algorithmes Evolutif, du Gradient et l'Aléatoire-4.

Le programme 23 est du type"fourier-par.prg" et possède un seul processus, qui est placé au centre de la machine et qui génère N d'autres processus ($N = 2^k$, $k >= 1$). Dans ce programme $N = 2^9 = 512$ processus. Le volume de données échangés = 3000um et la quantité de calculs effectués entre chaque échange = 15000uc. Les résultats obtenus utilisant une grille de taille 10x10 se trouve sur le tableau suivant:

Méthode	*Paramètres*	*Noeuds utilisés*	*Temps*	*Nb.processus sur chaque noeud*
Gradient	*S. Moyen* : 6 *S. Chargé* : 8	79	472300 *ut*	0 – 9
Aléatoire – 4	– – –	82	184680 *ut*	0 – 15
Evolutif	*nb.Processus/NvCh* : 1 *Taille max SP* : 6 *Validité SP* : 80	60	177457 *ut*	0 – 15

Nous pouvons voir sur le coin supérieur gauche de la figure 5.44, la distribution des processus obtenue par l'algorithme du Gradient. Cet algorithme a obtenu le meilleur équilibrage de la charge sur la grille, cependant à force de vouloir utiliser le plus de noeuds, le temps d'exécution a augmenté considérablement.
Le graphe situé sur le coin supérieur droite de la même figure montre la distribution des processus obtenue par l'algorithme Aléatoire-4 qui a pu réduire le temps d'exécution du programme. Dans ce cas on voit que l'algorithme Aléatoire-4 a utilisé presque le même nombre de noeuds que celui du Gradient, cependant la répartition des processus a été moins équitable chargeant surtout les noeuds du centre.
Finalement nous avons sur la partie basse de la figure 5.44 la distribution des processus donné par l'algorithme Evolutif. On peut voir que malgré la diminution du nombre de noeuds utilisés, la qualité de l'équilibrage de la charge obtenu se trouve entre celle donnée par la méthode du Gradient et celle de l'Aléatoire-4, donnant en plus, le temps d'exécution le plus petit.

Le deuxième exemple a pour but de comparer les performances des méthodes de placement dans le cas où plusieurs programmes s'exécutent sur la même machine.

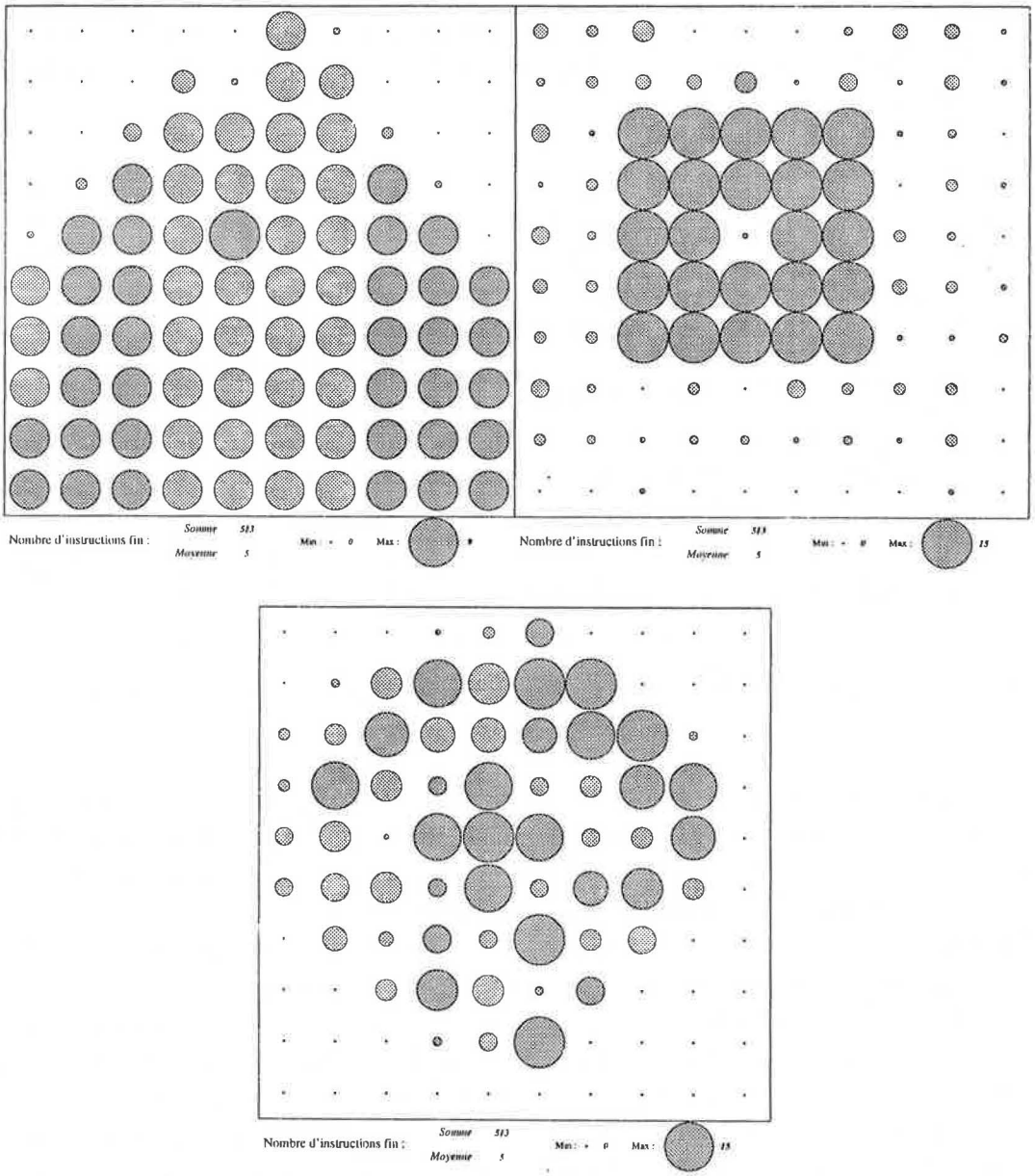


FIG. 5.44 - *Nombre de processus exécutés sur chaque processeur, Aléatoire-25 (en haut à gauche), Gradient (en haut à droite) et Evolutif (en bas)*

Pour cela nous avons choisi 4 programmes du type "espace-d-etats.prg" qui ont un graphe de créations en forme d'arbre représentant l'espace d'états du problème pour placer N reines sur un tableau de taille N. Les quatre processus initiaux des programmes ont été placés sur chacun des quatre coins d'une grille de taille 8x8.

Les variables des programme ont été initialisées de la manière suivante: nombre de reines à placer et la taille du tableau égales à 6, vol_comm = 250000um, nombre de communications entre père et fils = 2 et la quantité de calculs à réaliser = 200000uc. Le nombre de processus générés par chaque programme = 150, et le nombre total de processus à placer est donc 150x4 = 600 processus.

La table suivante montre les temps d'exécution obtenus par les algorithmes de placement: Aléatoire-4, Aléatoire-25, du Gradient et l'Evolutif.

Méthode	*Paramètres*	*Temps*	*Nb.processus sur chaque noeud*
Aléatoire − 4	− − −	2455280	0 − 31
Aléatoire − 25	− − −	2190359	1 − 22
Gradient	*S. Moyen* : 7 *S. Chargé* : 11	933727	0 − 11
Evolutif	*nb.Processus/NvCh* : 1 *Taille max SP* : 4 *Validité SP* : 500000	1020045	6 − 12

Dans ce test, où les processus initiaux des programmes se placent sur les quatre coins de la grille, l'algorithme Evolutif a occupé la deuxième place après la méthode du Gradient, ensuite les algorithmes Aléatoire-25 et l'Aléatoire-4. Les distributions des processus obtenues par les quatre algorithmes se trouvent sur la figure 5.45.

Dans ce cas, la méthode du Gradient distribue les processus vers les noeuds faiblement chargés du centre de la machine. C'est à dire que le problème de trouver une région de noeuds surchargés ne se pose pas immédiatement. D'autre part, après avoir étudié le comportement de l'algorithme Evolutif, on peut dire que son comportement est meilleur lorsque le placement démarre au centre de la machine. L'explication est que les SP pour actualiser l'information arrivent plus vite lorsqu'un noeud est au centre que lorsqu'il est au coin. Pour valider cette intuition on a fait exécuter les mêmes programmes mais avec les processus initiaux placés plus vers le centre de la grille (sur les positions (3,3), (3,6), (6,3) et (6,6)). Les résultats on confirmé notre

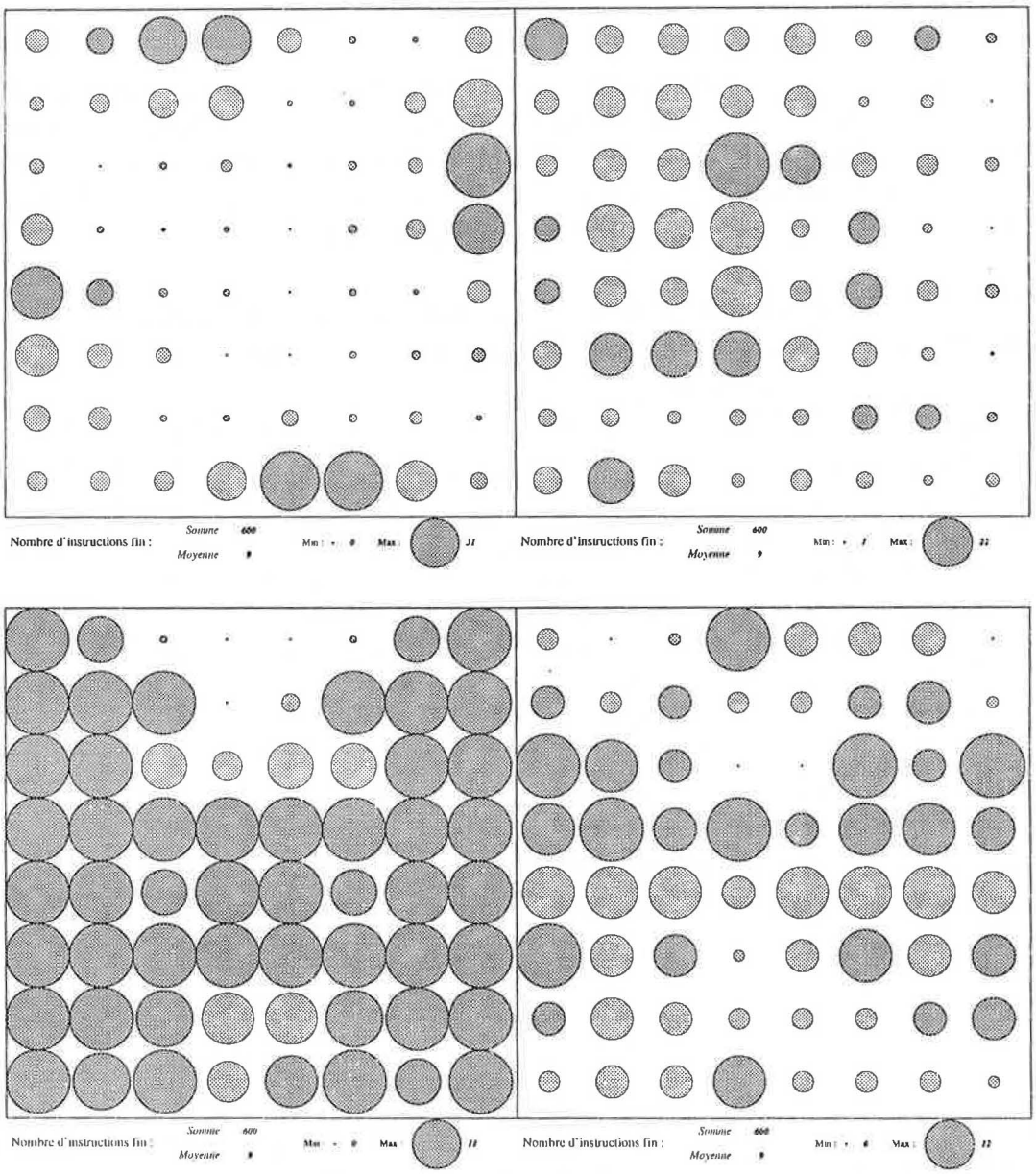


FIG. 5.45 - *Nombre de processus exécutés sur chaque processeur, Aléatoire-4, Aléatoire-25 (en haut), Gradient et Evolutif (en bas)*

intuition, la table suivante montre les temps d'exécution obtenus par les différentes méthodes.

Méthode	*Paramètres*	*Temps*	*Nb.processus sur chaque noeud*
Aléatoire – 4	– – –	3103146	0 – 29
Aléatoire – 25	– – –	1986636	1 – 24
Gradient	*S. Moyen* : 7 *S. Chargé* : 11	1386700	0 – 16
Evolutif	*nb.Processus/NvCh* : 1 *Taille max SP* : 4 *Validité SP* : 500000	1207201	4 – 13

On peut voir sur la figure 5.46 la distribution des processus pour les quatre méthodes de placement. Dans ce test l'algorithme Evolutif a donné le temps le plus petits d'exécution et la distribution des processus a été plus équitable qu'avec les autres algorithmes.

5.3.4 Comparaison obtenue en faisant varier les paramètres de SIMAD

De nombreux jeux de test ont été réalisés pour comparer l'algorithme Evolutif avec l'Aléatoire et celui du Gradient (quelques exemples se trouvent dans [RACS96a] et [PRAS95]). Dans cette dernière comparaison nous présentons les résultats obtenus après l'exécution de 4608 configurations de simulation. Les configurations utilisent 24 programmes de type 2 ("arbre-complet-1"), 3 ("prg-all-1.prg") et 12 ("arbre-complet-2") qui s'exécutent sur une grille de taille 12x12 et en utilisant différentes valeurs pour les paramètres du simulateur SIMAD.

La grille a les caractéristiques suivantes:

- Débit des liens: 10um/ut
- Capacité de calcul des noeuds: 1000uc/ut
- Capacité de mémoire des noeuds: 100000um

Pour les 24 programmes chaque processus crée au maximum 2 autres processus. Les programmes sont classés en trois groupes:

- premier groupe
  La quantité de calculs à réaliser est égale pour tous les processus.

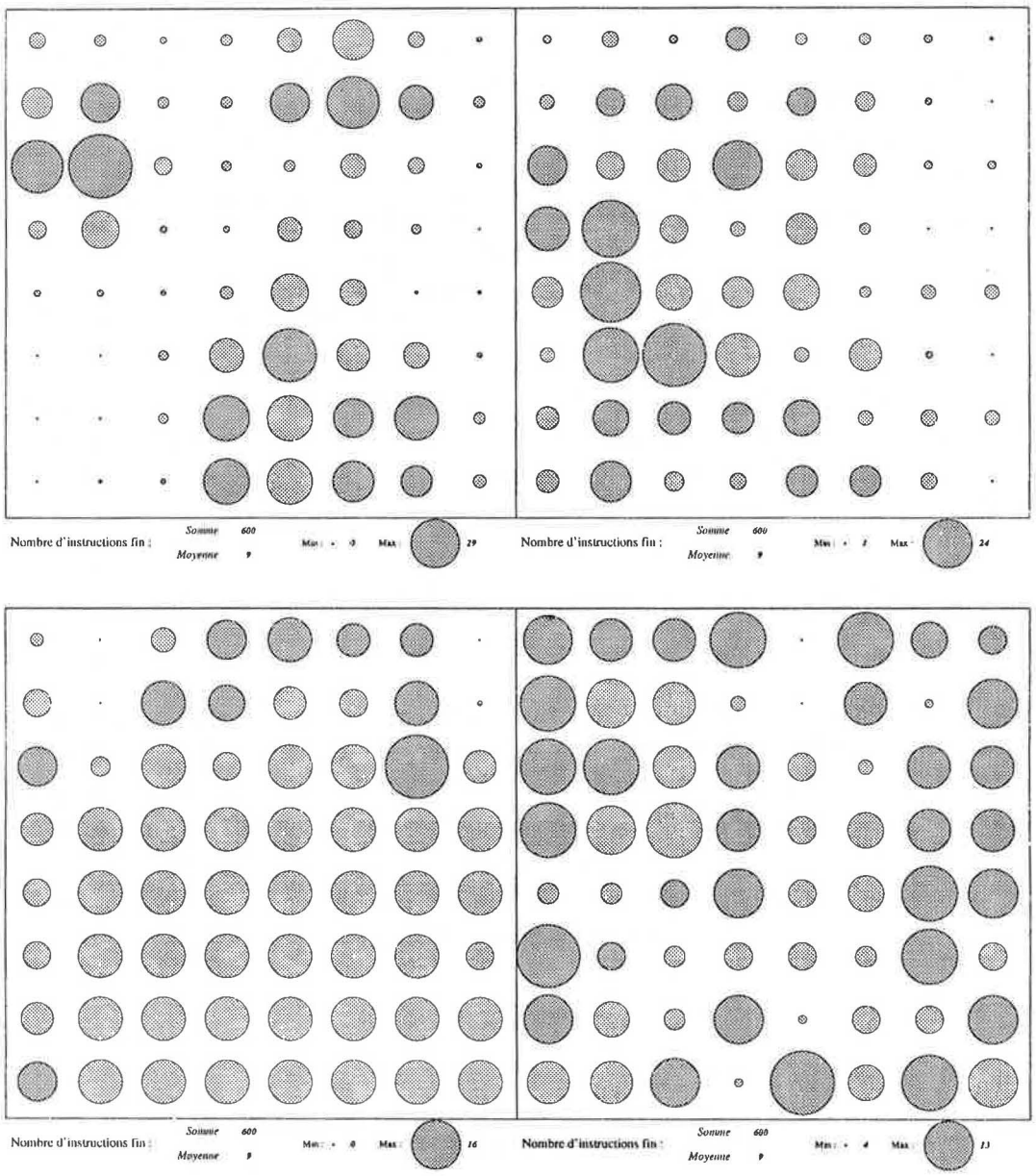


FIG. 5.46 - *Nb de processus exécutés sur chaque processeur, Aléatoire-4,Aléatoire-25 (en haut), Gradient et Evolutif (en bas)*

Il y a seulement un envoi de message d'un processus fils vers son père lorsque le processus fils finit son exécution. Chaque processus père crée ses processus fils, puis il fait des calculs, ensuite il reçoit les messages de leurs fils et finalement il envoi un message a son père.

- deuxième groupe
  La quantité de calculs à réaliser par chaque processus décroît selon le niveau qu'il a dans l'arbre de créations. Le processus racine fait plus de calculs que les autres noeuds.
  Comme dans le premier groupe, il y a seulement un envoi de message du processus fils vers le père.

- troisième groupe
  La quantité de calculs à réaliser par chaque processus décroît selon le niveau qu'il a dans l'arbre de créations. Le processus racine fait plus de calculs que les autres noeuds.
  Chaque père et fils font 2 échanges de messages. Entre chaque réception et envoi, les processus font des calculs. Un dernier message est envoyé du processus fils vers son père lorsque le processus fils finit son exécution.

Tous les programmes varient dans:

- La quantité de calculs effectuée par chaque processus
- La quantité de mémoire qu'ils utilisent
- La quantité de communications entre un processus père et son fils.

En ce qui concerne les paramètres de SIMAD, les options d'exécution ont été les suivantes:

- Type de partage du processeur: Equitable ou Prioritaire (on favorise les processus qui pourraient débloquer d'autres processus)
- Site de placement du processus initial: Sur le noeud 0 (un coin) ou bien au centre de la machine)
- Durée maximale d'occupation du processeur par un processus: 100ut, 1000ut, 10000ut et 100000ut
- Si aucun processus ne peut être exécuté, le noeud est mis en attente pendant 1000ut, 10000ut et 100000ut
- Pénalisation pour les messages qui transitent sur des noeuds qui ne sont pas les destinataires: 10ut, 100ut, 500ut et 1000ut

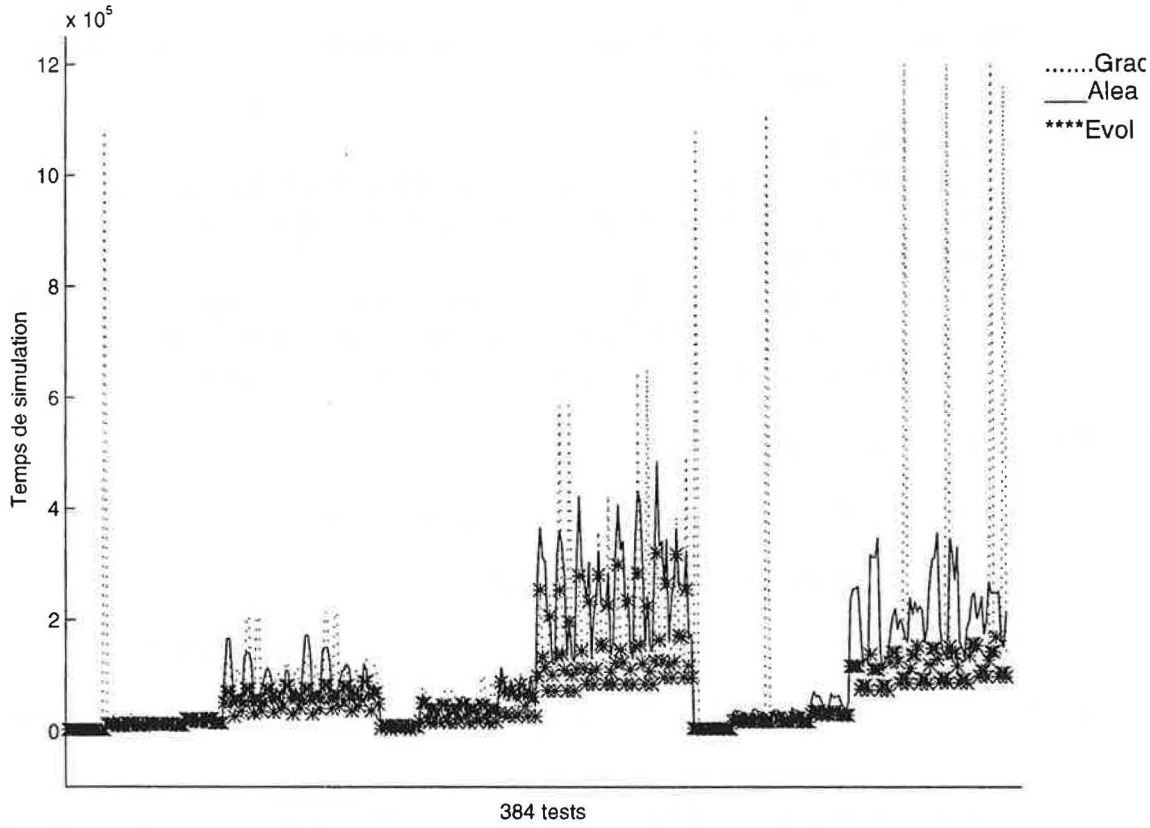


FIG. 5.47 - *Temps de simulation obtenus par les algorithmes, Aléatoire4, Gradient et Evolutif sur 384 tests*

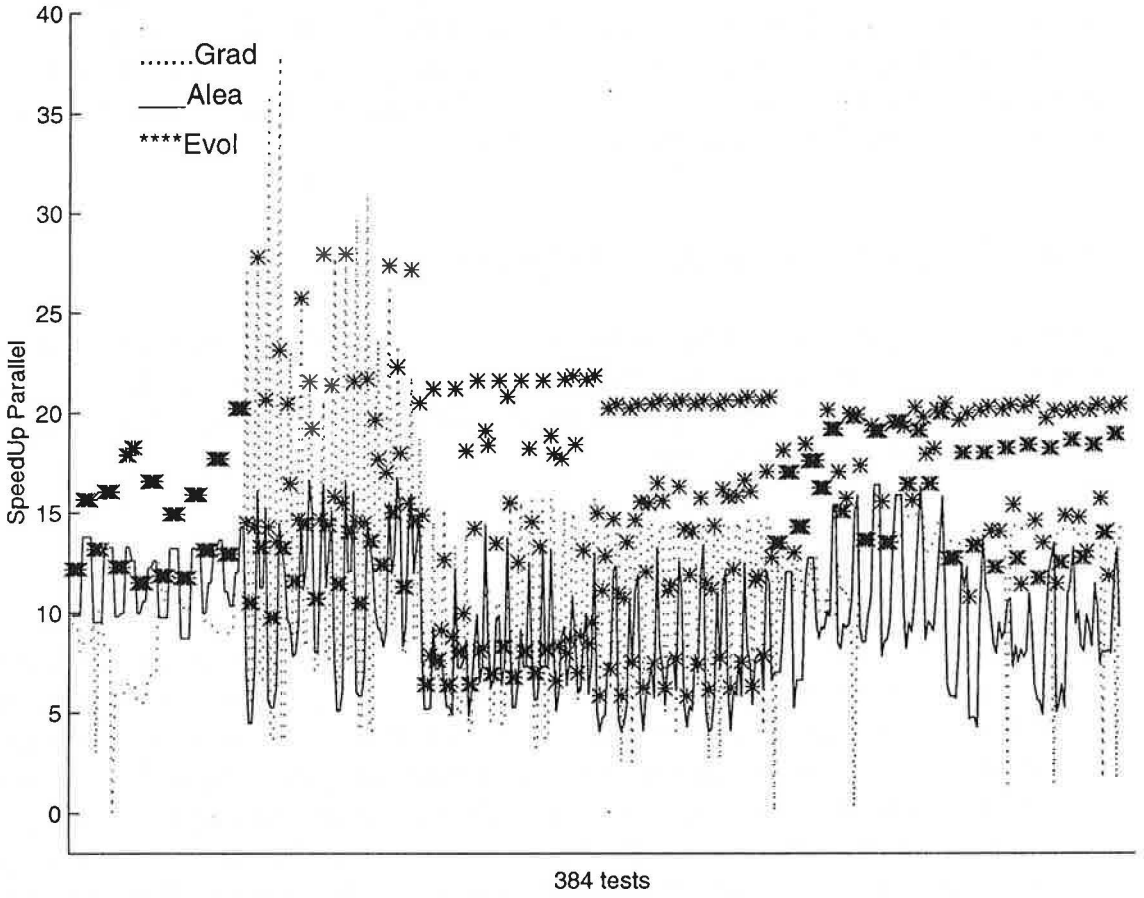


FIG. 5.48 - *Speedup obtenu par les algorithmes Aléatoire4, Gradient et Evolutif sur 384 tests*

Nous avons fait exécuter les algorithmes Aléatoire-4, du Gradient et l'Evolutif sous ces configurations de simulation. La figure 5.47 montre les temps de simulation obtenus par les trois algorithmes dans 384 tests. Ces tests ont été numérotés et choisis parmi la totalité (4608 tests) en sélectionnant tous les numéros multiples de 12. Sur la figure 5.48 on peut voir le speedup parallèle des mêmes 384 tests. D'une manière générale nous avons trouvé que l'algorithme Evolutif a eu des bonnes performances dans l'éxécution de ce jeux de tests.

5.4 Conclusions du chapitre

Dans une première partie de ce chapitre nous avons étudié le comportement particulier de l'algorithme Evolutif selon la variation de ses paramètres de base. Pour étudier l'influence d'un certain paramètre sur les performances nous l'avons fait varier en fixant les autres paramètres.

En résumé, le résultat de cette étude est le suivant:

- Différentes tailles maximales des SP peuvent donner des temps d'exécution variés. Après avoir réalisé de nombreux tests nous n'avons pas trouvé une règle pour déterminer quelle taille maximale donnera le temps d'exécution le plus petit. Cependant nous donnons un critère basé sur la taille de la machine utilisée et la taille du programme exécuté qui pourrait être pris en compte pour fixer la valeur maximale des SP. Un problème se pose lorsqu'on ne connaît pas les caractéristiques du programme, dans ce cas nous croyons qu'il est préférable d'utiliser des tailles de SP petites pour être sûr que la connaissance sur l'état de la machine pourra être actualisée plus fréquemment.

- Le temps maximum de validité d'une SP a été défini pour éviter le blocage de l'exécution d'un programme. Nous avons montré par un exemple comment ce problème peut arriver surtout dans le cas où les créations du programme sont centralisées. Si les créations sont distribuées sur plusieurs noeuds de la machine, l'évolution des SP est déclenché par les noeuds appliquant les opérateurs de déclin, croissance, fusion et remplacement. Pour les programmes où les créations sont centralisées les meilleurs performances ont été obtenues en utilisant des temps de validité petits pour actualiser plus fréquemment les SP

des noeuds. Si les créations sont distribuées sur la machine, les meilleurs performances sont obtenues en utilisant des temps de validité grands et laissant les SP s'actualiser par l'application des opérateurs évolutifs.

- La différence entre le niveau de charge associé à un noeud et le niveau de charge associé à sa SP détermine le site où les processus créés vont être placés. Le nombre de processus par niveau de charge d'un noeud a une influence directe sur l'évolution des SP. Si nous utilisons 1 processus par niveau de charge, les SP seront reconstruites fréquemment générant plus de messages sur la machine. De manière générale, ce numéro peut être fixé selon les objectifs que l'on cherche. Si l'objectif est d'utiliser le plus de noeuds de la machine, nous pouvons définir 1 ou 2 processus par niveau de charge. Cependant dans certains cas, l'utilisation de beaucoup de noeuds ne réduit pas forcement le temps d'exécution et il est préférable d'augmenter le nombre de processus par niveau de charge. Dans cette étude nous avons comparé deux types de niveaux de charge, le premier utilise un nombre maximum de processus qui est constant et le deuxième utilise un nombre maximum de processus qui s'adapte aux changements de la charge. Nos tests on montré que de manière générale le niveau de charge adaptable améliore les performances du niveau de charge constant. Nous faisons remarquer qu'il est possible d'utiliser d'autres mesures de la charge différentes de celle du nombre de processus sur un noeud. Au lieu d'utiliser de compteurs de processus nous pourrions, par exemple, utiliser la taille de la queue d'entrée/sortie.

- La façon de reconstruire les SP peut aussi influencer les performances de l'algorithme Evolutif. Nous avons comparé un premier type de SP formée par des noeuds de même niveau de charge avec un deuxième type de SP formée par des nocuds de différents niveau de charge mais ordonnées de manière croissante. Dans la plus grande partie des tests réalisés, les SP du premier type ont donnée les meilleures performances grâce à sa façon très pratique de reconstruction.

La deuxième partie de ce chapitre a montrée une comparaison de l'algorithme Evolutif avec les méthodes du Gradient, l'Aléatoire-4 et l'Aléatoire-25 exécutant les programmes définis dans le chapitre 3.10. Ces programmes sont destinés à tester les algorithmes de placement dynamique, ils sont divisés en 4 groupes. Les deux premiers groupes sont des programmes qui ne surchargent pas les liens de communication et diffèrent par le type de graphe de créations utilisé. Les deux derniers groupes sont des programmes qui génèrent plusieurs échanges de messages entre processus et diffèrent aussi par le type de graphe de créations utilisé.

Dans cette partie comparative nous n'avons pas cherché à optimiser le comportement des algorithmes Evolutif ni celui du Gradient, nous avons simplement initialisé

leurs paramètres de telle sorte qu'ils puissent utiliser le plus de noeuds sur la machine. Dans les tests effectués, l'algorithme Evolutif a utilisé des tailles petites des SP et un nombre petit de processus par niveau de charge assurant une fréquente actualisation de la connaissance. En prenant en compte le nombre total de processus des programmes, les seuils de la méthode du Gradient ont été fixés le plus petits possible pour utiliser le plus de noeuds de la machine. D'autre part, les algorithmes Aléatoire-4 et Aléatoire-25 n'ont pas de paramètres qui influencent leur comportement.

Pour évaluer les performances des algorithmes de placement nous avons pris en considération le temps d'exécution obtenu, le nombre de noeuds utilisés et l'équilibrage de la charge sur les noeuds. Après chaque comparaison, nous avons présenté quelques exemples particuliers où l'algorithme Evolutif a montré de bonnes performances. Les comparaisons effectuées ont montré que l'algorithme Evolutif arrive à avoir un bon comportement pour l'exécution des 4 groupes de programmes.

De manière générale, pour un programme donné, notre étude permet de voir (en faisant de simulations) comment régler les paramètres de l'algorithme Evolutif ainsi que de choisir le meilleur algorithme de placement dynamique pour exécuter ce programme. En faisant référence aux nombreuses expériences faites, il est possible aussi de se rapprocher d'un ou plusieurs programmes contenus dans les différents jeux de test pour prévoir le comportement de l'algorithme de placement et juger de son adéquation au placement des processus de ce programme.

Conclusion

Nous avons traité du problème du placement dynamique des processus d'un programme sur un système parallèle formé par des noeuds interconnectés. Afin d'avoir une connaissance globale du domaine, nous avons présenté un résumé des algorithmes de placement statique, utilisés lorsque les caractéristiques des programmes peuvent être estimées avant l'exécution.

Si le nombre de processus d'un programme varie au cours d'une exécution, il est préférable d'utiliser un algorithme de placement dynamique qui décide du placement des processus durant l'exécution. Nous avons présenté une recherche bibliographique des divers travaux réalisés sur les algorithmes de placement dynamique des processus. Parmi les travaux réalisés, certains présentent des classifications d'algorithmes de placement dynamique basées sur les caractéristiques de leurs éléments de contrôle et d'information, selon son caractère centralisé ou distribué. Nous avons proposé une nouvelle classification qui prend en compte le type d'information collectée dans les divers algorithmes.

Nous considérons que la transmission d'un message contenant un sous-ensemble d'états de charge de plusieurs noeuds (vecteur) est une bonne solution, car elle réduit le nombre de messages sur les liens et elle représente une information plus cohérente par rapport à l'état de charge réel de la machine. Cette solution, qui donne aux noeuds une connaissance partielle de l'état de charge du système est rarement envisagée, car la façon d'actualiser le vecteur n'est pas facile à déterminer.

Les outils spécialisés dans la mesure des performances des algorithmes de placement ne sont pas très nombreux. Nous en avons étudiés quelques uns ayant des caractéristiques différentes. L'outil SIMAD a été présenté en détail car il a été la plate-forme d'évaluation de notre travail.
Lorsque le but principal est de tester le comportement des algorithmes de placement dynamique, l'utilisation de programmes de type synthétique facilite la réalisation des objectifs. Une partie de cette thèse a été consacrée à la proposition et à l'implantation dans SIMAD d'un langage de haut niveau (LASSIMAD) qui permet d'écrire

des programmes de type synthétique pour tester des algorithmes de placement dynamique.
LASSIMAD utilise des instructions comme celles que le langage PVM utilise pour créer des processus, ainsi que pour envoyer et recevoir des messages. Une instruction spéciale permet d'occuper le processeur pendant un certain temps. Jusqu'à présent, l'implantation des opérations de diffusion et de réduction de messages utilisant des groupes de noeuds n'a pas encore été effectuée sur SIMAD.

Pour le test du langage LASSIMAD, nous avons construit deux groupes principaux de programmes pour tester les algorithmes de placement dynamique. Le premier groupe est composé par des programmes où les processus font plutôt des calculs que de communications. Dans le deuxième groupe, les processus font plusieurs échanges de messages dans leur exécution. Chacun des groupes a été divisé en deux sous-groupes selon le graphe de créations des programmes. Les charges générées par ces programmes n'ont pas une relation directe avec le travail fait dans un programme réel. Nous avons simplement défini certaines quantités de calcul et de communications expérimentales pour charger les noeuds et les liens de la machine.

Nous avons proposé un nouvel Algorithme Evolutif de placement dynamique de processus pour des machines de type MIMD à mémoire distribuée. L'Algorithme Evolutif donne à chaque noeud de la machine une connaissance partielle du système formée par un sous-ensemble de noeuds. On appelle ce sous-ensemble "la Solution de Placement" (SP) du noeud. La SP initiale d'un noeud a été définie comme l'ensemble de ses voisins directement connectés.

Nous avons consacré un des chapitre de cette thèse à l'explication du comportement de l'Algorithme Evolutif. Nous avons introduit la notion de niveau de charge qui se substitue à la notion de seuil utilisée par plusieurs algorithmes de placement dynamique pour équilibrer la charge sur la machine. Dans l'élément de contrôle de l'Algorithme Evolutif, les décisions de placement et d'acceptation d'un processus sont basées en prenant en compte le niveau de charge d'un noeud X par rapport au niveau de charge de sa SP.

La caractéristique la plus importante de l'Algorithme Evolutif est l'utilisation de plusieurs opérateurs (déclin, croissance, rotation, remplacement et fusion) pour actualiser les SP des noeuds selon les changements de la charge sur le système. Ces opérateurs permettent une reconstruction rapide des SP (utilisées dans n'importe quel moment de l'évolution).

Finalement, nous avons présenté une étude du comportement de l'Algorithme Evo-

lutif ainsi qu'une évaluation par comparaison avec la méthode du Gradient et deux algorithmes Aléatoires. Nos tests ont été basés sur l'exécution des deux groupes de programmes proposés pour tester les algorithmes de placement dynamique.

Dans l'étude du comportement de l'Algorithme Evolutif, nous avons fait varier ses paramètres de base (taille maximale des SP, temps maximum de validité des SP et nombre de processus par niveau de charge) pour étudier leur influence sur les performances en exécutant quelques programmes de test. Nous avons proposé certaines valeurs pour ces paramètres selon des caractéristiques générales des applications. Les performances de l'algorithme Evolutif ont été aussi étudiées utilisant deux types de niveaux de charge (adaptable et constant) et deux types de reconstruction des SP (avec des noeuds de même niveau de charge et avec des noeuds de niveaux de charge différents mais ordonnés de manière croissante).

Pour évaluer l'Algorithme Evolutif nous l'avons comparé avec la méthode du Gradient et deux algorithmes Aléatoires, en exécutant les programmes de test proposés sur plusieurs grilles de noeuds. Dans cette partie, notre objectif n'était pas d'optimiser les paramètres des algorithmes Evolutif et du Gradient pour exécuter au mieux les programmes. Notre objectif était plutôt de fixer les paramètres des algorithmes avec l'intention d'utiliser le plus de noeuds possibles de la machine. Nos résultats ont montré de meilleures performances de l'algorithme Evolutif dans la plupart des tests.

Cette thèse propose plusieurs perspectives de travail de domaines relatifs à l'étude du placement dynamique de processus :

Dans l'Algorithme Evolutif proposé, nous avons utilisé une seule SP par noeud. Nous pourrions utiliser plusieurs types de SP dans le cas où l'on travaille avec des noeuds ayant plusieurs caractéristiques ou rendant plusieurs services. De cette façon, on pourrait définir une SP_1 contenant des noeuds qui rendraient le service 1, une autre SP_2 contenant des noeuds qui rendraient le service 2, etc. Si on voulait placer un processus qui occupe un service i, on pourrait utiliser les noeuds dans SP_i. De cette façon, chaque SP dans un noeud pourrait évoluer de manière indépendante pour être actualisée.

Nous avons contribué à l'amélioration du type de programmes exécutés par SIMAD. Nous pourrions envisager d'améliorer la partie concernant les machines utilisées. La création d'un nouveau générateur de machines permettant de décrire différents types de topologies (comme l'hypercube, l'anneau, le tore ou les arbres) offrirait d'autres axes de recherche.

Une perspective envisageable serait de construire un outil qui reçoit un programme écrit dans un langage de programmation parallèle comme PVM et qui génère automatiquement les programmes synthétiques pour SIMAD. De cette manière, nous pourrions obtenir une estimation de l'exécution d'un programme réel en utilisant un algorithme de placement donné.

Nous pourrions également tester les performances estimées par SIMAD utilisant des vrais systèmes parallèles. Il pourrait être intéressant d'implanter l'algorithme Evolutif sur un système parallèle réel (des grilles ou d'autres types de topologies) dans le but de distribuer sa charge et de vérifier les performances diagnostiqués par le simulateur.

Finalement, nous pouvons dire que nous avons atteint nos objectifs en donnant de nouvelles perspectives d'étude sur la problématique du placement dynamique de processus et en proposant un outil d'aide à la mise au point et à l'évaluation des algorithmes de placement dynamique de processus sur des machines de type MIMD à mémoire distribuée.

Références

[AP88] André (F.) et Pazat (J-L.). – Le placement de tâches sur des architectures parallèles. *T.S.I.*, vol. 7, n° 4, 1988, pp. 385–401.

[BA86] Ben-Ari (M.). – *Processus concurrents, Introduction à la programmation parallèle.* – MASSON, 1986.

[Ban91] Banâtre (J.P.). – *La Programmation Parallèle, Outils, Méthodes, Eléments de mise en oeuvre.* – Eyrolles, Paris, 1991.

[BBS95] Bernon (C.), Betourne (C.) et Sayah (A.). – Placement dynamique de processus communicants par coopération entre agents. *In : 3èmes Journnées Francophones IAD & SMA.* – Chambery- St Baldoph.

[BCG87] Berry (G.), Couroné (P.) et Gonthier (G.). – Programmation synchrone des systèmes réactifs: le langage esterel. *T.S.I.*, no4, 1987, pp. 305–3316.

[BKM95] Briat (J.), Kannat (S.E.) et Morel (E.). – Plate-forme d'évaluation de stratégies de régulation dynamique de charge pour le système logique parallèle PLoSys. *In : RenPar'7*, éd. par Dekeyser (Libert et Manneback). – PIP-FPMs Mons, Belgique, May 1995.

[BM88] Bollinger (S.W.) et Midkiff (S.F.). – Processor and link assignment in multicomputers using simulated annealing. *In : International Conference on Parallel Processing*, pp. 1–7.

[Bok81a] Bokhari (S.H.). – On the mapping problem. *IEEE Transactions on Computers*, vol. C-30, n° 3, mars 1981, pp. 207–214.

[Bok81b] Bokhari (S.H.). – A shortest tree algorithm for optimal assignment across space and time in a distributed processor system. *IEEE Transactions on Software Engineering*, vol. SE-7, n° 6, novembre 1981, pp. 583–589.

[BS85] Barak (A.) et Shiloh (A.). – A distributed load–balancing policy for a multicomputer. *Software – Practice and Experience*, vol. 15, n° 9, septembre 1985, pp. 901–913.

[BSS91] Bernard (G.), Steve (D.) et Simatic (M.). – Placement et migration de processus dans les systèmes répartis faiblement couplés. *T.S.I.*, vol. 10, n° 5, 1991, pp. 355–373.

[BST89] Bal (H.E.), Steiner (J.G.) et Tanenbaum (A.S.). – Programming languages for distributed computing systems. *ACM Computing Surveys*, vol. 21, n° 3, September 1989, pp. 261–311.

[CC88] Carlier (J.) et Chrétienne (P.). – *Problème d'ordonnancement - modélisation / complexité / algorithmes.* – Masson, Paris, 1988.

[CC94] Calvin (C.) et Colombet (L.). – *Introduction au modèle de programmation par processus communicants : deux exemples PVM et MPI.* – Rapport Apache n° 12, Grenoble, IMAG - equipe APACHE, juillet 1994.

[CDJ+91] Covington (R.G.), Dwarkadas (S.), Jump (J.R.), Sinclair (J.B.) et Madala (S.). – The efficient simulation of parallel computer systems. *In : International Journal in Computer Simulation*, pp. 31–58.

[CF95] Corbel (A.) et Fleter (F.). – Linda: Un modèle de programmation parallèle. *Calculateurs Parallèles*, vol. 7, n° 2, 1995.

[Chr94] Christaller (M.). – *Athapascan-0a sur PVM3 : definition et mode d'emploi.* – Rapport Apache n° 11, Grenoble, IMAG - equipe APACHE, juin 1994.

[CRP96] Castaeda-Retiz (M.R.) et Plateau (B.). – Evaluation des stratégies de régulation dynamique de charge. *In : RenPar'8.* – Université de Bordeaux I, Talence, France, Mai 1996.

[CY90] Chen (G.H.) et Yur (J.S.). – A branch and bound with understimates algorithm for the task assignment problem with precedence constraint. *In : The 10th International Conference on Distributed Computer Systems*, pp. 494–501. – Paris, France.

[DR94] Dowaji (S.) et Roucairol (C.). – Load balancing strategy and priority of tasks in distributed environments. *In : Fourteenth Annual IEEE Conference- International Phoenix Conference on Computers and Communications*, pp. 15–22. – Scottsdale, Arizona(USA).

[Dzi90] Dzierzgowski (D.). – Quatre exemples de langages ou environnements pour le développement de programmes où le temps intervient. *T.S.I.*, no4, 1990, pp. 289–312.

[EKMT94] Elleuch (A.), Kanawati (R.), Muntean (T.) et Talbi (E-G.). – Dynamic load balancing mechanisms for a parallel operating system kernel. *In : LNCS.* – Springer Verlag. Linz, Austria.

[Ell94] Elleuch (A.). – *La migration de processus dans les systèmes parallèles.* – Thèse de PhD, INPG, Grenoble, Nov 1994.

[ELZ86] Eager (D.L.), Lazowska (E.D.) et Zahorjan (J.). – Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, vol. SE-12, n° 5, May 1986, pp. 662–675.

[EM94] Elleuch (A.) et Muntean (T.). – Process migration protocols for massively parallel systems. *In : MPCS.* – Ischia.

[Eti94] Etiemble (D.). – L'évolution des machines parallèles et massivement parallèles. *In : Ecole d'été parallélisme du CNRS.* – Lyon.

[Fol96] Folliot (B.). – *Contribution à une approche système du placement dynamique dans les systèmes répartis hétérogènes.* – Paris VI, Thèse de PhD, Université Pierre et Marie Curie, Dec 1996. Habilitation à diriger des recherches.

[Ga94] Geist (A.) et al. – *PVM 3 Users's guide and reference manual.* – Rapport technique n° ORNL/TM-12187, Oak Ridge, Tennessee 37831, Oak Ridge National Laboratory, September 1994. Engineering Physics and Mathematics Division.

[GJ79] Garey (M.R.) et Johnson (D.S.). – *Computers and Intractability. A Guide into the Theory of NP–Completeness.* – W. H. Freeman and Company, 1979.

[Gol89] Goldberg (D.E.). – *Genetic Algorithms in Search, Optimization, and Machine Learning.* – Addison – Wesley, 1989.

[gr:94] *PROTEUS Parallel Programming Language.* – University of North Carolina, 1994. Serveur: duke.cs.duke.edu.

[HAR94] Hou (E.S.), Ansari (N.) et Ren (H.). – A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, n° 2, February 1994, pp. 113–120.

[Hem94] Hemery (F.). – *Etude de la répartition dynamique d'activités sur architectures décentralisées.* – France, Thèse de PhD, Université des Sciences et Technologies de Lille, Juin 1994.

[Her93] Herrod (S.A.). – *Tango Lite: A multiprocessor simulation environment.* – Rapport technique, Computer systems laboratory, Standford university, 1993.

[Hoa78] Hoare (C.A.). – Communicating sequential processes. *Communications of the ACM*, vol. 21, n° 8, 1978, pp. 666–677. – Programming Techniques.

[INM88] INMOS. – *OCCAM 2 Reference Manual.* – INMOS, 1988.

[Isl94] Islam (N.). – *Customized Message Passing and Scheduling for Parallel and Distributed Applications.* – Thèse de PhD, Departement of Computer Science, University of Illinois at Urbana-Champaign, 1994. Report No. UIUCDCS-R-94-1856.

[Jég96] Jégou (Y.). – Exploitation du parallélisme à grain fin par migration de tâches sur les architectures à mémoire distribuées. *Calculateurs Parallèles*, vol. 8, n° 3, 1996, pp. 295–309.

[JM93] Jacqmot (C.) et Milgrom (E.). – A systematic approach to load distibution strategies for distributed systems. *In : Descentralized and Distributed Systems*, pp. 291–303.

[JM96] Jacqmot (C.) et Milgrom (E.). – Evaluation empirique des performances d'un système informatique: application à l'équilibrage de charge. *In : Ecole Placement Dynamique et Répartition de Charge*, éd. par PRS (GDR-PRC), pp. 127–139.

[JS96] Juganaru (M.) et Sakho (I.). – Répartition dynamique de données régulières pour des machines MIMD homogènes à mémoire distribuée. *Calculateurs parallèles*, vol. 8, n° 1, 1996, pp. 89–102.

[Jum93a] Jump (J.R.). – *NETSIM Reference Manual.* – ECE Dept., Rice University, May 1993. email jrj@rice.edu.

[Jum93b] Jump (J.R.). – *YACSIM Reference Manual.* – ECE Dept., Rice University, March 1993. email jrj@rice.edu.

[Kit94] Kitajima (J.P.). – *Modèles quantitatifs d'algorithmes parallèles.* – France, Thèse de PhD, Institut National Polytechnique de Granoble, Octobre 1994.

[KR91] Kumar (V.) et Rao (V.). – Scalability of parallel algoritms for the all-pairs shortest path problem : A summary of results. *Journal of Parallel and Distributed Computing*, vol. 13, 1991, pp. 124–138.

[LA87] Lee (S.Y.) et Aggarwal (J.K.). – A mapping strategy for parallel processing. *IEEE Transactions on Computers*, vol. 36, n° 4, April 1987, pp. 433–442.

[Len93] Lenstra (J.K.). – Approximation algorithms for job shop scheduling. *In : Workshop on Models and Algorithms for Planning and Scheduling Problems.*

[Ler80] Leroudier (J.). – *La simulation à événements discrets.* – Editions Hommes et Techniques, 1980.

[LK87] Lin (F.C.) et Keller (R.M.). – The gradient model load balancing method. *IEEE Transactions on Software Engineering*, vol. SE-13, n° 1, janvier 1987, pp. 32–38.

[Lo84] Lo (V.M.). – Heuristic algorithms of task assignment in distributed systems. *In : 4th International Conference on Distributed Computer Systems*, pp. 30–39.

[Lo88a] Lo (V.M.). – Algorithms for static task assignment and symetric contraction in distributed systems. *In : International Conference on Parallel Processing*, pp. 239–244.

[Lo88b] Lo (V.M.). – Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers*, vol. 37, n° 11, novembre 1988, pp. 1384–1397.

[Mar93] Marquet (P.). – Langages et expression du parallélisme de données. *T.S.I.*, vol. 12, n° 6, 1993.

[ME90] Macfarlane (D.) et East (I.). – An investigation of several parallel genetic algorithms. *In : Proc. of the 12th Occam User Group*, éd. par Exeter, pp. 60–67. – England.

[MLT82] Ma (P.Y.), Lee (E.Y.) et Tsuchiya (M.). – A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, vol. C-31, n° 1, January 1982, pp. 41–47.

[MM89] Magirou (V.F.) et Milis (J.Z.). – An algorithm for the multiprocessor assignement problem. *Operations Research Letters*, vol. 8, 1989, pp. 351–356.

[MT91a] Muntean (T.) et Talbi (E.-G.). – Methodes de placement statique des processus sur architectures paralleles. *T.S.I.*, vol. 10, n° 5, 1991, pp. 355–373.

[MT91b] Muntean (T.) et Talbi (E-G.). – A parallel genetic algorithm for process-processors mapping. *In : High Performance Computing II*, éd. par M.Durand et Dabaghi (F. El). pp. 71–82. – Elsevier Science Publishers B.V.(North-Holland).

[Neu91] Neuhaus (P.). – Solving the mapping-problem-experiences with a genetic algorithm. *In : In parallel problem Solving from Nature.* pp. 170–175. – Springer–Verlag.

[Pa93] Plateau (B.) et al. – *Présentation d'APACHE.* – Rapport APACHE n° 1, Grenoble, IMAG, octobre 1993.

[Pav93] Pavard (G.). – *Allocations de tâches sur machines MIMD à mémoire distribuée.* – Rapport technique, Heuristique et Diagnostic des Systèmes Complexes, Université de Technologie de Compiègne, Janvier 1993.

[Pop90] Poplawski (D.A.). – *Synthetic models of distributed memory parallel programs.* – Technical Report ORNL/Tm-11634, Oak Ridge, Tenessee 37831-USA, Oak Ridge National Laboratory- Martin Marietta, 1990.

[PRAS95] Pavard (G.), Román-Alonso (G.) et Sayettat (C.). – L'allocation dynamique sur machines MIMD avec le simulateur SIMAD. *In : Journées de Recherche sur le placement dynamique et la répartition de charge: Application aux systèmes répartis et parallèles.* – Paris, France, 1995.

[PTS88] Pulidas (S.), Towsley (D.) et Stankovic (J.A.). – Imbedding gradient estimators in load balancing algorithms. *In : 8th Int. Conf. on Distributed Computing Systems*, pp. 482–490. – San Jose, California, 1988.

[RACS96a] Román-Alonso (G.), Corbel (A.) et Sayettat (C.). – Etude comparative d'un algorithme évolutif pour l'équilibrage dynamique de charge. *In : Ecole Placement Dynamique et Répartition de charge.* – Presqu'île de Gien, France, Juillet 1996.

[RACS96b] Román-Alonso (G.), Corbel (A.) et Sayettat (C.). – Un algorithme évolutif pour l'équilibrage dynamique de charge de machines MIMD. *In : RenPar'8.* – Université de Bordeaux I, Talence, France, Mai 1996.

[RAS94] Román-Alonso (G.) et Sayettat (C.). – Evolution dynamique de la charge d'une machine parallèle basée sur des opérateurs génétiques. *In :*

Evolution Artificielle 94. pp. 133–141. – CEPADUES. ENAC Toulouse, France.

[RF94] Raverdy (P.-G.) et Folliot (B.). – Migration efficace d'applications parallèles au moyen des territoires d'exécution. *In : RenPar'6*, pp. 225–229. – ENS Lyon, France.

[RM91] Rakotoarisoa (H.) et Mussi (P.). – *PARSEVAL:PARallélisation sur réseaux de Transputers de Simulations pour l'EVALuation de performances.* – Rapport Technique n° 131, Sophia Antipolis, France, Unité de recherche INRIA-SOPHIA ANTIPOLIS, Septembre 1991.

[RVV92] Roch (J.L.), Vermeerbergen (A.) et Villard (G.). – Cost prediction for load-balancing : application to algebraic computations. *In : Parallel Processing: CONPAR92 - VAPP V*, pp. 467–478. – Lyon, France, 1992.

[SFWS92] Schnekenburger (T.), Friedrich (M.), Weininger (A.) et Schoen (T.). – Parsim: A tool for the analysis of parallel and distributed programs. *In : International Conference on Parallel Processing*, pp. 689–700. – Lyon.

[SG87] Suh (J.Y.) et Gucht (D.V.). – *Distributed Genetic Algorithms.* – Technical report no. 225, Bloomington, IN 47405, Computer Science Department, Indiana University, July 1987.

[Sin87] Sinclair (J.B.). – Efficient computation of optimal assignements for distributed tasks. *Journal of Parallel and Distributed Computing*, vol. 4, 1987, pp. 342–362.

[SK93] Stromboni (J.-P.) et Kwiatkowski (L.). – Conception d'un modèle pour l'analyse du parallélisme. *In : OPOPAC. Actes des Journées Internationales sur les Problèmes Fondamentaux de l'Informatique Parallèle et Distribuée*, éd. par Lavallée I. (Parker Y.), pp. 171–185.

[ST85] Shen (C.C.) et Tsai (W.H.). – A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, vol. C-34, n° 3, Mars 1985, pp. 197–203.

[Ste93] Stender (J.). – *Parallel Genetic Algorithms: Theory & Applications.* – IOS Press, 1993, *Frontiers in Artificial Intelligence and Applications.*

[Sto77] Stone (H.S.). – Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, vol. SE-3, n° 1, janvier 1977, pp. 85–93.

[Tal93] Talbi (E.-G.). – *Allocation de processus sur les architestures parallèles a mémoire distribueé.* – Thèse de PhD, INPG, 1993.

[Tal95] Talbi (E.-G.). – *Allocation dynamique de processus dans les systèmes distribués et parallèles: Etat de l'art.* – Rapport technique No. 162, Laboratoire d'Informatique Fondamentale de Lille, 1995.

[TBAM96] Talbi (E-G.), Bessière (P.), Ahuactzin (J.) et Mazer (E.). – Un algorithme génétique parallèle pour l'optimisation. *T.S.I.*, vol. 15, n° 8, 1996, pp. 1105–1130.

[TL89] Theimer (M.M.) et Lantz (K.A.). – Finding idle machines in a worksation-based distributed system. *IEEE transactions on Software Engineering*, vol. 15, n° 11, november 1989, pp. 1444–1458.

[US90] Udiavar (N.) et Stiles (G.S.). – A simple but flexible model for determining optimal task allocation and configuration on a network of transputers. *In : NATUGI: Transputer Research and Applications*, pp. 24–32. – Utah, USA.

[WM93] Woodside (C.M.) et Monforton (G.G.). – Fast allocation of processes in distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, n° 2, February 1993, pp. 164–174.

[XH91] Xu (J.) et Hwang (K.). – Mapping rule-based systems onto multicomputers using simulated annealing. *Journal of Parallel and Distributed Computing*, vol. 13, 1991, pp. 442–454.

[Yah94] Yahfoufi (N.). – *Comparaison de SIMAD : SImulateur de Méthodes d'Allocation Dynamique, avec d'autres simulateurs de machines parallèles.* – Rapport technique, Université de Technologie de Compiegne, Département de Génie Informatique, Septembre 94. Rapport de DEA Contrôle des Systèmes.

[YSM95] Yan (J.), Sarukkai (S.) et Mehra (P.). – Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software – Practice and Experience*, vol. 25, n° 4, April 1995, pp. 429–461.

[Zho88] Zhou (S.). – A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, vol. 14, n° 9, September 1988, pp. 1327–1341.

Annexe A

Les programmes de test où il y a peu de communications

A.1 Programme 1.- processus-indep.prg

```
main
memory = 1000;
var son_number,tree_height,i,id,quant_comp,nb_total_processus;
{
  quant_comp = 50;
  son_number = 2;
  tree_height = 10;
  nb_total_processus = 1023;
  id= spawn(TREE, tree_height, son_number, quant_comp,mytid);
  for(i=1;i<= nb_total_processus;i=i+1)
     recv(nktid,data);
}

process TREE(tree_height, son_number, quantity_comp,id_racine)
memory = 1000;
var i,id[5],height, nb_recep;
{
   if(tree_height > 1)
   {
      height = tree_height - 1;
      for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,height,son_number,quantity_comp,id_racine);
      compute(quantity_comp);
   }
   else
      compute(quantity_comp);
   send(id_racine,data,1000);
}
```

A.2 Programme 2.- arbre-complet-1.prg

```
main
memory = 1000;
```

```
var son_number,tree_height,i,id,quant_comp;
{
  quant_comp = 10000;
  son_number = 5;
  tree_height = 5;
  id= spawn(TREE, tree_height, son_number, quant_comp);
  recv(id,nktype);
}

process TREE(tree_height, son_number, quantity_comp)
memory = 1000;
var i,id[5],height, nb_recep;
{
   if(tree_height > 1)
   {
      height = tree_height - 1;
      for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,height,son_number,quantity_comp);
      compute(quantity_comp);
      for(i= 0; i< son_number; i=i+1)
        recv(nktid,data);

   }
   else
      compute(quantity_comp);
   send(parent,data,1000);
}
```

A.3 Programme 3.- prg-all-1.prg

```
main
memory = 1000;
var son_number,tree_height,son_height,id[4],quant_comp,
    quant_comm,son_comput,i,calc_NvSup;
{
  quant_comp = 1800000;
  quant_comm = 1000;
  son_number = 4;
  tree_height = 5;
  calc_NvSup = 250;

  if(tree_height > 1)
  {
      son_height = tree_height - 1;
      son_comput = quant_comp - calc_NvSup;
       for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,son_height,son_number,son_comput,
                                        quant_comm,calc_NvSup);
      compute(quant_comp);
      for(i= 0; i< son_number; i=i+1)
        recv(nktid,data);
   }
   else
      compute(quant_comp);
}

process TREE(tree_height, son_number, quantity_comp,quant_comm,calc_NvSup)
memory = 1000;
var i,id[4],son_height,son_comput;
{
```

```
   if(tree_height > 1)
   {
      son_height = tree_height - 1;
      son_comput = quantity_comp - calc_NvSup;
       for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,son_height,son_number,
                              son_comput,quant_comm,calc_NvSup);
      compute(quantity_comp);
      for(i= 0; i< son_number; i=i+1)
        recv(nktid,data);

   }
   else
      compute(quantity_comp);
   send(parent,data,quant_comm);
}
```

A.4 Programme 4.- pparallel.prg

```
main
memory = 1000;
var son_number,tree_height,i,id[4],quant_comp,quant_comp_feuille,
    height, capacite_calc_machine,
    quant_comp_envoi_tache,quant_comp_arrive_mess,
    temps_transfert_tache,temps_transfert_donnees,temps_creation,
    quant_comp_inst_creation,quant_comp_inst_recoit;
{
  quant_comp_feuille = 100000;
  son_number = 3;
  tree_height = 7;
  capacite_calc_machine = 1000;
  temps_transfert_tache = 100;
  temps_transfert_donnees = 1000;
  temps_creation = 6;

   if(tree_height > 1)
   {
      height = tree_height - 1;
      for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,height,son_number,
quant_comp_feuille,capacite_calc_machine,
               temps_transfert_tache,temps_transfert_donnees,temps_creation);

      if( height > 1)
      {
        quant_comp_inst_creation = son_number * capacite_calc_machine;
        quant_comp_inst_creation = quant_comp_inst_creation * temps_creation;
        quant_comp_inst_recoit = son_number * capacite_calc_machine;
      }
      else
      {
        quant_comp_inst_creation = son_number * capacite_calc_machine;
        quant_comp_inst_creation = quant_comp_inst_creation * temps_creation;
        quant_comp_inst_recoit = 0;
      }
      quant_comp_envoi_tache = temps_transfert_tache * capacite_calc_machine;
      quant_comp_arrive_mess = temps_transfert_donnees * capacite_calc_machine;

      quant_comp = quant_comp_envoi_tache + quant_comp_arrive_mess;
      quant_comp = quant_comp + quant_comp_inst_creation;
```

```
      quant_comp = quant_comp + quant_comp_inst_recoit;
      quant_comp = quant_comp * height;

      quant_comp = quant_comp + quant_comp_feuille;

      compute(quant_comp);
      for(i= 0; i< son_number; i=i+1)
        recv(nktid,data);

   }
   else
      compute(quant_comp_feuille);
}

process TREE(tree_height, son_number, quant_comp_feuille,
          capacite_calc_machine,temps_transfert_tache,temps_transfert_donnees,
          temps_creation)
memory = 1000;
var i,id[4],height,nb_recep,
    quant_comp_inst_creation, quant_comp_inst_recoit,quant_comp_envoi_tache,
    quant_comp_arrive_mess,quant_comp;
{
   if(tree_height > 1)
   {
      height = tree_height - 1;
      for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,height,son_number,
quant_comp_feuille,capacite_calc_machine,
 temps_transfert_tache,temps_transfert_donnees,temps_creation);

      if( height > 1)
      {
        quant_comp_inst_creation = son_number * capacite_calc_machine;
        quant_comp_inst_creation = quant_comp_inst_creation * temps_creation;
        quant_comp_inst_recoit = son_number * capacite_calc_machine;
      }
      else
      {
        quant_comp_inst_creation = son_number * capacite_calc_machine;
        quant_comp_inst_creation = quant_comp_inst_creation * temps_creation;
        quant_comp_inst_recoit = 0;
      }

      quant_comp_envoi_tache = temps_transfert_tache * capacite_calc_machine;
      quant_comp_arrive_mess = temps_transfert_donnees * capacite_calc_machine;

      quant_comp = quant_comp_envoi_tache + quant_comp_arrive_mess;
      quant_comp = quant_comp + quant_comp_inst_creation;
      quant_comp = quant_comp + quant_comp_inst_recoit;
      quant_comp = quant_comp * height;

      quant_comp = quant_comp + quant_comp_feuille;

      compute(quant_comp);
      for(i= 0; i< son_number; i=i+1)
        recv(nktid,data);
   }
   else
      compute(quant_comp_feuille);
   send(parent,data,10000);
}
```

A.5 Programme 5.- travaille-tjrs.prg

```
main
memory = 1000;
var son_number,tree_height,son_height,id[4],i,feuille_comput,
quant_comp,quant_comm,nb_reponses,test;
{
  feuille_comput = 2800000;
  quant_comp = 50000;
  quant_comm = 1000;
  son_number = 3;
  tree_height = 6;

  if(tree_height > 1)
  {
      son_height = tree_height - 1;
      for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,son_height,son_number,
quant_comp,quant_comm,feuille_comput);
      for(nb_reponses= 0; nb_reponses< son_number; i=i+1)
      {
        test = nrecv(nktid,data);
        if(test == 1)
        {
recv(nktid,data);
nb_reponses = nb_reponses + 1;
}
else
       compute(quant_comp);
      }
  }
  else
      compute(quant_comp);
}

process TREE(tree_height, son_number, quant_comp,quant_comm,feuille_comput)
memory = 1000;
var i,id[4],son_height,test,nb_reponses;
{
   if(tree_height > 1)
   {
      son_height = tree_height - 1;
      for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,son_height,son_number
,quant_comp,quant_comm,feuille_comput);
      for(nb_reponses= 0; nb_reponses< son_number; i=i+1)
      {
        test = nrecv(nktid,data);
        if(test == 1)
        {
recv(nktid,data);
nb_reponses = nb_reponses + 1;
}
else
       compute(quant_comp);
      }
   }
   else
      compute(feuille_comput);
   send(parent,data,quant_comm);
}
```

A.6 Programme 6.- arbr1-AuDiAuDi.prg

```
main
memory = 8000;
var son_number,tree_height,son_height,id[3],quant_comp,quant_comm,
    son_comput,i,calc_NvSup;
{
  quant_comp = 2500000;
  quant_comm = 10000;
  son_number = 3;
  tree_height = 7;
  calc_NvSup = 170000;

  if(tree_height > 1)
  {
      son_height = tree_height - 1;
      son_comput = quant_comp - calc_NvSup;
      for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,son_height,son_number,son_comput,
                    quant_comm,calc_NvSup);
      compute(quant_comp);
      for(i= 0; i< son_number; i=i+1)
        recv(nktid,data);

      for(i= 0; i< son_number; i=i+1)
            id[i] = spawn(TREE,son_height,son_number,son_comput,quant_comm);
      compute(quant_comp);
      for(i= 0; i< son_number; i=i+1)
            recv(nktid,data);
   }
   else
      compute(quant_comp);
}

process TREE(tree_height, son_number, quantity_comp,quant_comm,calc_NvSup)
memory = 8000;
var i,id[3],son_height,son_comput;
{
   if(tree_height > 1)
   {
      son_height = tree_height - 1;
      son_comput = quantity_comp - calc_NvSup;
      for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,son_height,son_number,son_comput,
                                              quant_comm,calc_NvSup);
      compute(quantity_comp);
      for(i= 0; i< son_number; i=i+1)
        recv(nktid,data);

   }
   else
      compute(quantity_comp);
   send(parent,data,quant_comm);
}
```

A.7 Programme 7.- p1pere-Nfils.prg

```
main
memory = 5000;
```

```
var nb_processus,i,id[150],quant_calc;
{
  quant_calc = 800000;
  nb_processus = 150;
  for(i = 0; i < nb_processus; i=i+1)
        id[i] = spawn(FILS, quant_calc);
  for(i = 0; i < nb_processus; i=i+1)
        recv(nktid,data);
}

process FILS(quantite_calc)
memory = 5000;
{
   compute(quantite_calc);
   send(parent,data,1000);
}
```

A.8 Programme 8.- p1pere-Nfils-sync.prg

```
list messages
 TSYNC

main
memory = 5000;
var nb_processus,i,id[128],quant_calc;
{
  quant_calc = 800000;
  nb_processus = 128;
  for(i = 0; i < nb_processus; i=i+1)
        id[i] = spawn(FILS, quant_calc);
  for(i = 0; i < nb_processus; i=i+1)
        recv(nktid,TSYNC);
  for(i = 0; i < nb_processus; i=i+1)
        send(id[i],TSYNC);
  for(i = 0; i < nb_processus; i=i+1)
        recv(nktid,data);
}

process FILS(quantite_calc)
memory = 5000;
{
   send(parent,TSYNC);
   recv(parent,TSYNC);
   compute(quantite_calc);
   send(parent,data,2000);
}
```

A.9 Programme 9.- disques.prg

```
list messages
   TFIN, REPONSE

main
memory = 30000;
var  id;
{
   id = spawn(debut);
```

```
   recv(id,TFIN);
}

process debut()
memory = 10000;
var i, j, k, long,longueur,var_aux, flag,rep, id[8], calc,
   valeur,tab1[8],tab2[8],tab3[8],tab4[8],somme[8];
{
   flag = 0;

   longueur = 6;
   valeur =  21;

   tab1[0] = 1;
   tab1[1] = 2;
   tab1[2] = 3;
   tab1[3] = 4;
   tab1[4] = 5;
   tab1[5] = 6;

   tab2[0] = 1;
   tab2[1] = 2;
   tab2[2] = 3;
   tab2[3] = 4;
   tab2[4] = 5;
   tab2[5] = 6;

   tab3[0] = 1;
   tab3[1] = 2;
   tab3[2] = 3;
   tab3[3] = 4;
   tab3[4] = 5;
   tab3[5] = 6;

   tab4[0] = 1;
   tab4[1] = 2;
   tab4[2] = 3;
   tab4[3] = 4;
   tab4[4] = 5;
   tab4[5] = 6;

   for(i=0; i<longueur; i=i+1)
       somme[i] = tab1[i];
   for(i=0; i<longueur; i=i+1)
   {
       if(somme[i] > valeur)
   flag = 1;
   }
   if(flag == 1)
send(parent,TFIN);
   else
   {
for(i=0; i< longueur; i=i+1)
{
   var_aux= tab1[0];
      long = longueur - 1;
   for(j=0 ; j < long ; j=j+1)
     {
k = j + 1;
tab1[j] = tab1[k];
somme[j] = tab1[j];
   }
```

```
   tab1[long] = var_aux;
   somme[long] = tab1[long];
   id[i]=spawn(niveau1,tab2,tab3,tab4,somme,valeur,longueur);
}
calc = 0;
        for(i=0; i< longueur; i=i+1)
{
   recv(nktid,REPONSE,rep);
           if(rep > 0)
calc = calc + rep;
}
compute(calc);
send(parent,TFIN);
   }
}


process niveau1(tab1[8],tab2[8],tab3[8],somme[8],valeur,longueur)
memory = 10000;
var i, j, k, long, var_aux, flag,rep,somme2[8],nb_creations,id[8],calc;
{

   nb_creations = 0;

   for(i=0; i< longueur; i=i+1)
   {
       flag = 0;
           somme2 = somme;
   var_aux= tab1[0];
      long = longueur - 1;
   for(j=0 ; j < long ; j=j+1)
     {
k = j + 1;
tab1[j] = tab1[k];
        somme2[j] = somme2[j] + tab1[j];
   }
   tab1[long] = var_aux;
   somme2[long] = somme2[long] + tab1[long];
       flag = 0;
       for(j=0; j < longueur; j=j+1)
       {
        if(somme2[j] > valeur)
flag = 1;
   }
   if(flag == 0)
   {
   id[i]=spawn(niveau2,tab2,tab3,somme2,valeur,longueur);
           nb_creations = nb_creations + 1;
   }
   }
   flag = 0;
   for(i=0; i< nb_creations ; i=i+1)
   {
   recv(nktid,REPONSE,rep);
   if(rep > 0)
   {
         flag = flag + rep;
   }
   }
   if(flag > 0 )
      send(parent,REPONSE,flag);
   else
```

```
      send(parent,REPONSE,0);
}

process niveau2(tab1[8],tab2[8],somme[8],valeur,longueur)
memory = 10000;
var i, j, k, long, var_aux, flag,rep,calc,id[8],nb_creations,somme2[8];
{
   nb_creations = 0;

   for(i=0; i< longueur; i=i+1)
   {
          somme2 = somme;
   var_aux= tab1[0];
      long = longueur - 1;
   for(j=0 ; j < long ; j=j+1)
     {
k = j + 1;
tab1[j] = tab1[k];
        somme2[j] = somme2[j] + tab1[j];
   }
   tab1[long] = var_aux;
   somme2[long] = somme2[long] + tab1[long];
       flag = 0;
       for(j=0; j<longueur; j=j+1)
       {
        if(somme2[j] > valeur)
flag = 1;
   }
   if( flag == 0)
   {
   id[i]=spawn(niveau3,tab2,somme2,valeur,longueur);
          nb_creations = nb_creations + 1;
}
   }
   flag = 0;
   for(i=0; i< nb_creations ; i=i+1)
   {
   recv(nktid,REPONSE,rep);
   if(rep > 0)
   {
         flag = flag + rep;
   }
   }
   if(flag > 0)
      send(parent,REPONSE,flag);
   else
      send(parent,REPONSE,0);
}


process niveau3(tab1[8],somme[8],valeur,longueur)
memory = 10000;
var i, j, k, long, var_aux, flag,rep,calc,id[8],nb_solutions,somme2[8];
{
   nb_solutions = 0;
   for(i=0; i< longueur; i=i+1)
   {
          somme2 = somme;
   var_aux= tab1[0];
      long = longueur - 1;
   for(j=0 ; j < long ; j=j+1)
     {
```

```
k = j + i;
tab1[j] = tab1[k];
        somme2[j] = somme2[j] + tab1[j];
   }
   tab1[long] = var_aux;
   somme2[long] = somme2[long] + tab1[long];
       flag = 1;
       for(j=0; j<longueur; j=j+1)
       {
        if(somme2[j] > valeur)
flag = 0;
   }
   if(flag == 1)
      nb_solutions = nb_solutions + 1;
   }
   var_aux=mytid;
   compute(var_aux);
   send(parent,REPONSE,nb_solutions);
}
```

A.10 Programme 10.- reine.prg

```
list messages
   TFIN, REPONSE

main
memory = 1000;
var  id;
{
   id = spawn(debut);
   recv(id,TFIN);
}

process debut()
memory = 5000;
var i, niveau, niveau_suivant, taille_tab,quant_calc,solution[16],index,
                                                    id[16],rep,nb_rep,idp;
{
   niveau = 0;
   taille_tab = 4;
   nb_rep = 0;

   for(i=1; i<= taille_tab; i=i+1)
   {
solution[0] = i;
quant_calc = i * 100000;
index = i - 1;
       idp=i;
niveau_suivant = niveau + 1;
id[index] = spawn(reine,niveau_suivant,solution,
quant_calc,quant_calc,taille_tab,idp);
   }
   for(i=1; i<= taille_tab; i=i+1)
   {
   recv(nktid,REPONSE,rep,idp);
           if(rep > 0)
           {
        quant_calc = idp * 100000;
  compute(quant_calc);
              nb_rep = nb_rep + rep;
```

```
          }
   }
   compute(nb_rep);
   send(parent,TFIN);
}

process reine(niveau,solution[16],quant_calc,quant_orig,taille_tab,idp)
memory = 5000;
var i, j, flag,rep,nb_creations,id[16], position, niveau_suivant, index,
    diff_niveau, diag_gauche, diag_droite,quant_calc_suivant,nb_solutions ;
{

   nb_creations = 0;
   nb_solutions = 0;
   niveau_suivant = niveau + 1;
   for(j=1; j<= taille_tab; j=j+1)
   {
       flag = 0;
     for(i=0 ; i < niveau ; i=i+1)
     {
diff_niveau = niveau - i;
diag_gauche = solution[i] - diff_niveau;
diag_droite = diff_niveau + solution[i];
if( j == solution[i] ) flag = 1;
if( j == diag_gauche ) flag = 1;
if( j == diag_droite ) flag = 1;
   }
   if(flag == 0)
   {
solution[niveau] = j;
position = j * 1000;
quant_calc_suivant = quant_orig + niveau;
quant_calc_suivant = quant_calc_suivant + 1;
quant_calc_suivant = quant_calc_suivant + position;
if(niveau_suivant < taille_tab)
{
index = j - 1;
  id[index]=spawn(reine,niveau_suivant, solution,
quant_calc_suivant,quant_orig,taille_tab,idp);
         nb_creations = nb_creations + 1;
}
else
           {
compute(quant_calc_suivant);
nb_solutions = nb_solutions + 1;
}
   }
   }
   if(niveau_suivant < taille_tab)
   {
flag = 0;
    for(i=0; i< nb_creations ; i=i+1)
    {
   recv(nktid,REPONSE,rep,idp);
   if(rep > 0)
   {
          compute(quant_calc_suivant);
               nb_solutions = nb_solutions + rep;
flag = 1;
   }
    }
    if(flag == 1 )
```

```
        send(parent,REPONSE,nb_solutions,idp);
    else
        send(parent,REPONSE,0,idp);
   }
   else
   {
if(nb_solutions > 0)
            send(parent,REPONSE,nb_solutions,idp);
    else
        send(parent,REPONSE,0,idp);
   }
}
```

A.11 Programme 11.- arbre-tableau.prg

```
main
memory = 1000;
var index_son_number,tree_height,i,id[10],tab[5],quant_comp,
    quant_comp_feuille, son_number,
    height, capacite_calc_machine,
    quant_comp_envoi_tache,quant_comp_arrive_mess,
    temps_transfert_tache,temps_transfert_donnees,temps_creation,
    quant_comp_inst_creation,quant_comp_inst_recoit;
{
  tab[0]=7; tab[1]=6; tab[2]=5; tab[3]=2; tab[4]=3;
  quant_comp_feuille = 10000;
  son_number = tab[0];
  tree_height = 6;
  capacite_calc_machine = 1000;
  temps_transfert_tache = 100;
  temps_transfert_donnees = 1000;
  temps_creation = 6;

   if(tree_height > 1)
   {
      height = tree_height - 1;
      index_son_number = 1;
      for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,height,index_son_number,tab,
quant_comp_feuille,capacite_calc_machine,
               temps_transfert_tache,temps_transfert_donnees,temps_creation);

      if( height > 1)
      {
        quant_comp_inst_creation = son_number * capacite_calc_machine;
        quant_comp_inst_creation = quant_comp_inst_creation * temps_creation;
        quant_comp_inst_recoit = son_number * capacite_calc_machine;
      }
      else
      {
        quant_comp_inst_creation = son_number * capacite_calc_machine;
        quant_comp_inst_creation = quant_comp_inst_creation * temps_creation;
        quant_comp_inst_recoit = 0;
      }
      quant_comp_envoi_tache = temps_transfert_tache * capacite_calc_machine;
      quant_comp_arrive_mess = temps_transfert_donnees * capacite_calc_machine;

      quant_comp = quant_comp_envoi_tache + quant_comp_arrive_mess;
      quant_comp = quant_comp + quant_comp_inst_creation;
      quant_comp = quant_comp + quant_comp_inst_recoit;
```

```
      quant_comp = quant_comp * height;

      quant_comp = quant_comp + quant_comp_feuille;

      compute(quant_comp);
      for(i= 0; i< son_number; i=i+1)
        recv(nktid,data);

   }
   else
      compute(quant_comp_feuille);
}

process TREE(tree_height,index_son_number,tab[5],quant_comp_feuille,
             capacite_calc_machine,temps_transfert_tache,
             temps_transfert_donnees,temps_creation)
memory = 1000;
var i,id[10],height,nb_recep, son_number,quant_comp_inst_creation,
    quant_comp_inst_recoit,quant_comp_envoi_tache,
    quant_comp_arrive_mess,quant_comp;
{
   if(tree_height > 1)
   {
      son_number=tab[index_son_number];
      index_son_number = index_son_number + 1;

      height = tree_height - 1;
      for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,height,index_son_number,tab,
                 quant_comp_feuille,capacite_calc_machine,
 temps_transfert_tache,temps_transfert_donnees,temps_creation);

      if( height > 1)
      {
        quant_comp_inst_creation = son_number * capacite_calc_machine;
        quant_comp_inst_creation = quant_comp_inst_creation * temps_creation;
        quant_comp_inst_recoit = son_number * capacite_calc_machine;
      }
      else
      {
        quant_comp_inst_creation = son_number * capacite_calc_machine;
        quant_comp_inst_creation = quant_comp_inst_creation * temps_creation;
        quant_comp_inst_recoit = 0;
      }

      quant_comp_envoi_tache = temps_transfert_tache * capacite_calc_machine;
      quant_comp_arrive_mess = temps_transfert_donnees * capacite_calc_machine;

      quant_comp = quant_comp_envoi_tache + quant_comp_arrive_mess;
      quant_comp = quant_comp + quant_comp_inst_creation;
      quant_comp = quant_comp + quant_comp_inst_recoit;
      quant_comp = quant_comp * height;

      quant_comp = quant_comp + quant_comp_feuille;

      compute(quant_comp);
      for(i= 0; i< son_number; i=i+1)
        recv(nktid,data);
   }
   else
      compute(quant_comp_feuille);
   send(parent,data,10000);
```

}

Annexe B

Les programmes de test où les processus font plusieurs échanges

B.1 Programme 12.- arbre-complet-2.prg

```
main
memory = 1000;
var son_number,tree_height,i,j,id,quant_comp,vol_comm,nb_comm;
{
  quant_comp = 100000;
  vol_comm = 10000;
  son_number = 4;
  tree_height = 6;
  nb_comm = 0;

  id= spawn(TREE, tree_height, son_number, quant_comp,vol_comm,nb_comm);
  compute(quant_comp);
  for(j=0; j< nb_comm; j=j+1)
  {
    recv(id,data);
    compute(quant_comp);
    send(id,data,vol_comm);
  }
  recv(id,data);
}

process TREE(tree_height, son_number, quantity_comp,vol_comm,nb_comm)
memory = 1000;
var i,j,id[4],height;
{
   if(tree_height > 1)
   {
      height = tree_height - 1;
      for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,height,son_number,quantity_comp,vol_comm,nb_comm);
      compute(quantity_comp);
      for(j=0; j< nb_comm; j=j+1)
      {
          send(parent,data,vol_comm);
          compute(quantity_comp);
          for(i= 0; i< son_number; i=i+1)
```

```
            recv(nktid,data);
        recv(nktid,data);
        compute(quantity_comp);
        for(i= 0; i< son_number; i=i+1)
            send(id[i],data,vol_comm);
    }
    for(i= 0; i< son_number; i=i+1)
      recv(nktid,data);

  }
  else
  {
    compute(quantity_comp);
    for(j=0; j< nb_comm; j=j+1)
    {
        send(parent,data,vol_comm);
        compute(quantity_comp);
        recv(parent,data);
        compute(quantity_comp);
    }
  }
  send(parent,data,vol_comm);
}
```

B.2 Programme 13.- prg-comm-1.prg

```
main
memory = 1000;
var son_number,tree_height,son_height,i,id[4],
    quant_comp,vol_comm,son_comput,nb_comm,j,calc_NvSup;
{
  quant_comp = 1800000;
  vol_comm = 1000;
  nb_comm = 4;
  son_number = 4;
  tree_height = 5;
  calc_NvSup = 250;

  if(tree_height > 1)
  {
    son_height = tree_height - 1;
    son_comput = quant_comp - calc_NvSup;
     for(i= 0; i< son_number; i=i+1)
      id[i] = spawn(TREE,son_height,son_number,
                          son_comput,vol_comm,nb_comm,calc_NvSup);
    for(j=0; j< nb_comm; j=j+1)
    {
        compute(quant_comp);
        for(i= 0; i< son_number; i=i+1)
            recv(nktid,data);
        compute(quant_comp);
        for(i= 0; i< son_number; i=i+1)
            send(id[i],data,vol_comm);
    }
    for(i= 0; i< son_number; i=i+1)
      recv(nktid,data);
  }
  else
    compute(quant_comp);
}
```

```
process TREE(tree_height, son_number, quantity_comp,vol_comm,
                                                    nb_comm,calc_NvSup)
memory = 1000;
var i,j,id[4],son_height,son_comput;
{
   if(tree_height > 1)
   {
      son_height = tree_height - 1;
      son_comput = quantity_comp - calc_NvSup;
      for(i= 0; i< son_number; i=i+1)
        id[i] = spawn(TREE,son_height,son_number,son_comput,
                                                  vol_comm,nb_comm,calc);
      for(j=0; j<nb_comm; j=j+1)
      {
         compute(quantity_comp);
         send(parent,data,vol_comm);
         for(i= 0; i< son_number; i=i+1)
           recv(nktid,data);
         compute(quantity_comp);
         recv(nktid,data);
         for(i= 0; i< son_number; i=i+1)
           send(id[i],data,vol_comm);
      }
      for(i= 0; i< son_number; i=i+1)
        recv(nktid,data);
    }
   else
   {
      for(j=0; j<nb_comm; j=j+1)
      {
         compute(quantity_comp);
         send(parent,data,vol_comm);
         compute(quantity_comp);
         recv(parent,data);
      }
   }
   send(parent,data,vol_comm);
}
```

B.3 Programme 14.- comm3FFH7.prg

```
list messages
 TYPESYNC, TYPE1

main
memory = 1000;
var son_number,nb_comm_frere,id,quantity_comp,vol_comm, tree_height;
{
  quantity_comp = 150000;
  vol_comm = 80000;
  son_number = 3;
  tree_height = 7;
  nb_comm_frere = 3;
  id = spawn(TREE,tree_height,quantity_comp,vol_comm,son_number,nb_comm_frere);
  recv(id,TYPESYNC);
}


process TREE(tree_height, quantity_comp,vol_comm,son_number,nb_comm_frere)
```

```
memory = 1000;
var height,i,computation,id[3];
{
   if(tree_height > 1)
   {
     height = tree_height - 1;
     computation = quantity_comp - 12000;
     for(i = 0; i < son_number; i=i+1)
       id[i] = spawn(SON,height,son_number,computation,vol_comm,nb_comm_frere);
     for(i = 0; i < son_number; i=i+1)
        recv(nktid,TYPESYNC);
     for(i = 0; i < son_number; i=i+1)
        send(id[i],TYPE1,id,i,son_number);
     compute(quantity_comp);
     for(i = 0; i < son_number; i=i+1)
        recv(nktid,data);
   }
   send(parent,TYPESYNC);
}

process SON(tree_height,son_number,quantity_comp,vol_communication,nb_comm_frere)
memory = 1000;
var i,j,brothers[3],sons[3],myindex, brother_number,height,
computation,nb_recep;
{
   if(tree_height < 2)
   {
      send(parent, TYPESYNC);
      recv(parent, TYPE1, brothers, myindex, brother_number);
      for(j=0; j< nb_comm_frere; j=j+1)
      {
        for(i=0; i< brother_number; i=i+1)
     if( myindex != i)
  {
         compute(quantity_comp);
             send(brothers[i],data,vol_communication);
    }
        nb_recep = brother_number - 1;
        for(i=0; i< nb_recep; i=i+1)
          recv(nktid,data);
      }
      send(parent,data,vol_communication);
   }
   else
   {
     send(parent, TYPESYNC);
     height = tree_height - 1;
     computation = quantity_comp - 12000;
     for(i = 0; i < son_number; i=i+1)
      sons[i] = spawn(SON, height, son_number, computation,
                        vol_communication,nb_comm_frere);
     for(i = 0; i < son_number; i=i+1)
        recv(nktid,TYPESYNC);
     for(i = 0; i < son_number; i=i+1)
        send(sons[i],TYPE1,sons,i,son_number);
     recv(parent, TYPE1, brothers, myindex, brother_number);

     for(j=0; j< nb_comm_frere; j=j+1)
     {
       for(i=0; i< brother_number; i=i+1)
if( myindex != i)
{
```

```
        compute(quantity_comp);
            send(brothers[i],data,vol_communication);
    }
        nb_recep = brother_number - 1;
        for(i=0; i< nb_recep; i=i+1)
          recv(nktid,data);
      }
      for(i = 0; i < son_number; i=i+1)
         recv(nktid,data);
      send(parent, data,vol_communication);
    }
}
```

B.4 Programme 15.- espace-d-etats.prg

```
list messages
   TFIN, REPONSE

main
memory = 100;
var  id;
{
   id = spawn(debut);
   recv(id,TFIN);
}

process debut()
memory = 1000;
var i, niveau, niveau_suivant, taille_tab,calculs,solution[10],
    index, id[10],rep,vol_comm,j,nb_comm,nb_solutions,calculs_fils,calc_NvSup;
{
   taille_tab = 6;
   vol_comm = 25000;
   nb_comm = 2;
   calculs = 200000;
   calc_NvSup = 12000;

   niveau = 0;
   nb_solutions = 0;
   calculs_fils = calculs - calc_NvSup;
   for(i=1; i<= taille_tab; i=i+1)
   {
solution[0] = i;
index = i - 1;
niveau_suivant = niveau + 1;
id[index] = spawn(reine,niveau_suivant,solution,
calculs_fils,taille_tab,vol_comm,nb_comm,calc_NvSup);
   }

   for(j=0; j< nb_comm; j=j+1)
   {
     compute(calculs);
     for(i=0; i< taille_tab; i=i+1)
   recv(nktid,data);
     compute(calculs);
     for(i=0; i< taille_tab; i=i+1)
   send(id[i],data,vol_comm);
   }
   for(i=1; i<= taille_tab; i=i+1)
   {
```

```
   recv(nktid,REPONSE,rep);
   if(rep > 0) nb_solutions = nb_solutions + rep;
   }
   nb_solutions = nb_solutions * 1000;
   compute(nb_solutions);
   send(parent,TFIN);
}


process reine(niveau,solution[10],quant_calc,taille_tab,vol_comm,
            nb_comm,calc_NvSup)
memory = 1000;
var i, j, rep,nb_creations,id[10], niveau_suivant, index,flag,
    diff_niveau, diag_gauche, diag_droite,nb_solutions,quant_calc_fils ;
{
   quant_calc_fils = quant_calc - calc_NvSup;
   nb_creations = 0;
   nb_solutions = 0;
   niveau_suivant = niveau + 1;
   for(j=0; j < taille_tab; j=j+1)
       id[j] = -1;
   for(j=1; j<= taille_tab; j=j+1)
   {
       flag = 0;
     for(i=0 ; i < niveau ; i=i+1)
     {
diff_niveau = niveau - i;
diag_gauche = solution[i] - diff_niveau;
diag_droite = diff_niveau + solution[i];
if( j == solution[i] ) flag = 1;
if( j == diag_gauche ) flag = 1;
if( j == diag_droite ) flag = 1;
   }
   if(flag == 0)
   {
solution[niveau] = j;
if(niveau_suivant < taille_tab)
{
index = j - 1;
  id[index]=spawn(reine,niveau_suivant, solution,
  quant_calc_fils,taille_tab,
vol_comm,nb_comm);
         nb_creations = nb_creations + 1;
}
else
           {
compute(quant_calc);
nb_solutions = nb_solutions + 1;
}
   }
   }
   if(niveau_suivant < taille_tab)
   {
     for(j= 0; j < nb_comm; j=j+1)
      {
   send(parent,data,vol_comm);
           compute(quant_calc);
         recv(nktid,data);
           compute(quant_calc);
   if(nb_creations > 0)
           {
for(i= 0; i< nb_creations ; i=i+1)
```

```
          recv(nktid,data);
           compute(quant_calc);
           for(i =0; i< taille_tab ; i=i+1)
        { if(id[i] > 0 )
  send(id[i],data,vol_comm);
}
   }
        }
    for(i=0; i< nb_creations ; i=i+1)
    {
   recv(nktid,REPONSE,rep);
   if(rep > 0) nb_solutions = nb_solutions + rep;
    }
        send(parent,REPONSE,nb_solutions);
   }
   else
   {
       for(i= 0; i< nb_comm; i=i+1)
       {
           compute(5000);
   send(parent,data,vol_comm);
           compute(5000);
       recv(parent,data);
       }
       send(parent,REPONSE,nb_solutions);
   }
}
```

B.5 Programme 16.- fourier.prg

```
list messages
   TIDENT, TSYNCH

main
memory = 30000;
var nb_processus,i,id[1024],idf[10],num_id,gauche,
milieu,droite,degree,idindex,m,g,d,j,quant_comm,quant_calc;
{
  nb_processus = 1024;
  degree = 10;
  quant_comm = 80;
  quant_calc = 15000;

  gauche = 1;
  droite = nb_processus;
  milieu = gauche + droite;
  milieu = milieu / 2;
  compute(60);
  for(i = 0; i < nb_processus; i=i+1)
        id[i] = spawn(Fourier, degree,quant_comm,quant_calc);
  for(i = 0; i < nb_processus; i=i+1)
     recv(nktid,TSYNCH);
  compute(80);
  m=milieu;
  g=gauche;
  d=droite;
  for(i = 1; i <= nb_processus; i=i+1)
  {
        milieu=m;
gauche=g;
```

```
droite=d;
compute(90);
        for(j=0 ; droite > gauche ; j=j+1)
        {
    if(i <= milieu)
  {
      idindex = milieu + i;
   idindex = idindex - gauche;
           idf[j]=id[idindex];
   droite = milieu;
   milieu = milieu + gauche;
   milieu = milieu / 2;
   compute(100);
  }
  else
  {
      idindex = droite - i;
   idindex = milieu - idindex;
   idindex = idindex - 1;
           idf[j]=id[idindex];
   gauche = milieu + 1;
   milieu = droite + milieu;
   milieu = milieu / 2;
   compute(110);
  }
        }
      num_id = i - 1;
      send(id[num_id],TIDENT,idf);
  }
}

process Fourier(degree,quant_comm,quant_calc)
memory = 8000;
var i,id[10];
{
   send(parent,TSYNCH);
   recv(parent,TIDENT,id);
   for(i=0;i<degree;i=i+1)
   {
compute(quant_calc);
send(id[i],data,quant_comm);
   recv(id[i],data);
   }
}
```

B.6 Programme 17.- Fourier parallèle

```
list messages
   TIDENT, TSYNCH

main
memory = 8000;
var nb_processus,i,id[512],num_id,gauche,milieu,droite,degree,nbcomm,num,
      quant_calculs,vol_comm;
{
  nb_processus = 512;
  degree = 9;
  vol_comm = 3000;
  quant_calculs = 120000;
```

```
   gauche = 1;
   droite = nb_processus;
   milieu = gauche + droite;
   milieu = milieu / 2;
   for(i = 0; i < nb_processus; i=i+1)
   {
      num_id = i+1;
      id[i] = spawn(Fourier, num_id, gauche,milieu,droite,degree,
                    quant_calculs,vol_comm);
   }
   for(i = 0; i < nb_processus; i=i+1)
     recv(nktid,TSYNCH);
   nbcomm = droite * degree;
   for(i = 0; i < nbcomm; i=i+1)
   {
       recv(nktid,TIDENT,num_id);
       num_id = num_id -1;
       num = id[num_id];
       send(nktid,TIDENT,num);
   }
}

process Fourier(myid,gauche,milieu,droite,degree,quant_calculs,vol_comm)
memory = 8000;
var i,idreq,idindex,id[10];
{
   send(parent,TSYNCH);
   for(i=0 ; droite > gauche ; i=i+1)
   {
  if(myid <= milieu)
{
      idindex = milieu + myid;
   idindex = idindex - gauche;
   idindex = idindex + 1;
   send(parent,TIDENT,idindex);
   recv(parent,TIDENT,idreq);
           id[i]=idreq;
   droite = milieu;
   milieu = milieu + gauche;
   milieu = milieu / 2;
   compute(110);
}
else
{
      idindex = droite - myid;
   idindex = milieu - idindex;
   send(parent,TIDENT,idindex);
   recv(parent,TIDENT,idreq);
           id[i]=idreq;
   gauche = milieu + 1;
   milieu = droite + milieu;
   milieu = milieu / 2;
   compute(110);
}
   }
   for(i=0;i<degree;i=i+1)
   {
compute(quant_calculs);
send(id[i],data,vol_comm);
   recv(id[i],data);
   }
}
```

B.7 Programme 18.- prodN-consM.prg

```
list messages
typeSynch, typeReq, typeContinue, typeFin

main
memory = 2000;
var id[10],i,num_fils,nb_consommateurs,nb_produits_cons,nb_producteurs,
    nb_produits_fabr,temps_consomm,temps_product,nb_syst_prod_cons,
    volume_produit;
{
  num_fils = 2;
  nb_syst_prod_cons= 1;

  nb_consommateurs = 10;
  nb_produits_cons = 30;
  temps_product = 10000;

  nb_producteurs = 10;
  nb_produits_fabr = 30;
  temps_consomm = 10000;

  volume_produit = 100;

  for(i=0;i< num_fils;i=i+1)
    id[i] = spawn(crea_prod_cons,nb_syst_prod_cons,nb_consommateurs,
                nb_produits_cons,temps_product,nb_producteurs,
                nb_produits_fabr,temps_consomm,volume_produit);
  for(i=0;i< num_fils;i=i+1)
    recv(nktid,typeSynch);
}

process crea_prod_cons(num_fils,nb_consommateurs,nb_produits_cons,
                    temps_product,nb_producteurs,nb_produits_fabr,
                    temps_consomm,volume_produit)
memory = 2000;
var id[10],id2[10],i;
{
  for(i=0;i< num_fils;i=i+1)
    id[i] = spawn(prod1_consN,nb_consommateurs,nb_produits_cons,
                                          temps_product,volume_produit);
  for(i=0;i< num_fils;i=i+1)
    id2[i] = spawn(prodN_cons1,nb_producteurs,nb_produits_fabr,
                                          temps_consomm,volume_produit);
  for(i=0;i< num_fils;i=i+1)
    recv(nktid,typeSynch);
  for(i=0;i< num_fils;i=i+1)
    recv(nktid,typeSynch);
  send(parent,typeSynch);
}

process prod1_consN(nb_consommateurs,nb_produits_cons,temps_product,
                    volume_produit)
memory = 6000;
var idcons[50],prod,i,nb_produits_fabr,temps_consomm;
{
  send(parent,typeSynch);
  nb_produits_fabr = nb_produits_cons * nb_consommateurs;
  temps_consomm = temps_product * nb_consommateurs;

  prod = spawn(Producteur,nb_produits_fabr,temps_product,volume_produit);
  recv(prod,typeSynch);
```

```
  for(i=0;i< nb_consommateurs;i=i+1)
    idcons[i] = spawn(Consommateur,prod,nb_produits_cons,temps_consomm);
  for(i=0;i< nb_consommateurs;i=i+1)
     recv(nktid,typeSynch);
}

process prodN_cons1(nb_producteurs,nb_produits_fabr,temps_consomm,
                    volume_produit)
memory = 6000;
var idprod[50],cons,i,nb_produits_cons,temps_product;
{
  send(parent,typeSynch);
  nb_produits_cons = nb_produits_fabr * nb_producteurs;
  temps_product = temps_consomm * nb_producteurs;

  for(i=0;i< nb_producteurs;i=i+1)
    idprod[i] = spawn(Producteur,nb_produits_fabr,temps_product,volume_produit);
  for(i=0;i< nb_producteurs;i=i+1)
     recv(nktid,typeSynch);
  cons = spawn(Consommateur,idprod[0],nb_produits_cons,temps_consomm);
  recv(cons,typeSynch);
  for(i=1;i< nb_producteurs;i=i+1)
    send(idprod[i],typeReq,cons);
}

process Producteur(nb_produits,vitesse_product,volume_produit)
memory = 8000;
var i,idconsreq,nb_fabric;
{
    send(parent,typeSynch);
    nb_fabric = 0;
    for (i=0; i < nb_produits; i=i+1)
    {
      compute(vitesse_product);
      recv(nktid,typeReq,idconsreq);
      send(idconsreq,data,volume_produit);
      nb_fabric = nb_fabric + 1;
      if(nb_fabric < nb_produits)
         send(idconsreq,typeContinue);
      else
         send(idconsreq,typeFin);
    }
}

process Consommateur(idProd,nb_produits,vitesse_consom)
memory = 8000;
var i,id, nb_consom;
{
   send(parent,typeSynch);
   nb_consom = 0;
   send(idProd,typeReq,mytid);
   for (i=0; i < nb_produits; i=i+1)
   {
      recv(nktid,data);
      id = nktid;
      nb_consom = nb_consom + 1;
      recv(id,nktype);
      if(nb_consom < nb_produits)
      {
         if(nktype == typeContinue)
           send(id,typeReq,mytid);
      }
```

```
        compute(vitesse_consom);
    }
}
```

RESUME EN FRANCAIS :

Cette thèse est une contribution à l'étude du placement dynamique de processus sur des machines multiprocesseurs à mémoire distribuée. Le contexte de notre travail est celui de la simulation et de l'évaluation de l'exécution d'applications dont le nombre de processus et le moment de leur création dépendent de l'exécution en cours.

Nous proposons un nouvel algorithme Evolutif de placement dynamique de processus de type approximatif, avec des éléments de contrôle et d'information distribués. A chaque noeud X d'une machine parallèle est associé un sous-ensemble de processeurs avec lesquels il peut partager sa charge de manière équitable. Ce sous-ensemble est appelé la Solution de Placement (SP) du noeud. La Solution de Placement initiale d'un noeud X est composée du sous-ensemble des noeuds directement connectés au noeud X. La décision de placement d'un processus est faite au moment de sa création, il peut alors être placé sur le noeud sur lequel il a été créé ou bien sur un des noeuds de sa SP. Sous l'effet de certains opérateurs (déclin, croissance, fusion, remplacement, rotation) la Solution de Placement d'un noeud évolue au cours de l'exécution de l'application ce qui permet une répartition et un équilibrage des charges des noeuds.

Pour étudier le comportement de l'algorithme Evolutif, nous avons utilisé le simulateur séquentiel SIMAD qui est un outil conçu pour évaluer les algorithmes d'allocation dynamique de charge sur des machines MIMD à mémoire distribuée. Le deuxième apport de cette thèse est la définition et l'intégration dans SIMAD d'un langage synthétique qui permet de décrire des applications parallèles avec des graphes de communication généraux.

Le document se termine par la présentation d'une partie des résultats de l'ensemble des expériences que nous avons menées, dans le but d'évaluer les performances et le comportement de notre approche du placement dynamique de processus. Deux types de résultats sont présentés et analysés. Tout d'abord nous recherchons l'influence de certains paramètres (la taille maximale des SP, l'actualisation des SP, le nombre de processus par niveau de charge et l'opérateur de fusion) sur le comportement de l'algorithme Evolutif. Ensuite, une étude comparative avec d'autres méthodes de placement dynamique permet de mettre en évidence les performances de notre approche.

MOTS-CLES :

Algorithmes de placement dynamique
Machines parallèles
Modélisation de programmes parallèles
Evaluation par simulation